Lecture Notes in Computer Science 1576

S. Doaitse Swierstra (Ed.)

Programming Languages and Systems

8th European Symposium on Programming, ESOP'99
Held as Part of the Joint European Conferences
on Theory and Practice of Software, ETAPS'99
Amsterdam, The Netherlands, March 1999
Proceedings





Springer

Lecture Notes in Computer Science 1576
Edited by G. Goos, J. Hartmanis and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Singapore
Tokyo

S. Doaitse Swierstra (Ed.)

Programming Languages and Systems

8th European Symposium on Programming, ESOP'99
Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'99
Amsterdam, The Netherlands, March 22-28, 1999
Proceedings



Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editor

S. Doaitse Swierstra
Utrecht University, Department of Computer Science
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
E-mail: swierstra@cs.uu.nl

Cataloging-in-Publication data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Programming languages and systems : proceedings / 8th European Symposium on Programming, ESOP '99, held as part of the Joint European Conferences on Theory and Practice of Software, ETAPS '99, Amsterdam, The Netherlands, March 22 - 28, 1999. S. Doaitse Swierstra (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 1999
(Lecture notes in computer science ; Vol. 1576)
ISBN 3-540-65699-5

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

ISSN 0302-9743
ISBN 3-540-65699-5 Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

© Springer-Verlag Berlin Heidelberg 1999
Printed in Germany

Typesetting: Camera-ready by author
SPIN: 10703074 06/3142 – 5 4 3 2 1 0 Printed on acid-free paper

Preface

This is the second time that of ESOP has formed part of the ETAPS cluster of conferences, workshops, working group meetings and other associated activities. One of the results of colocating so many conferences is a reduction in the number of possibilities to submit a paper to a European conference and the increased competition between conferences that occurs when boundaries between individual conferences have not yet become well established. This may have been the reason for the fact that only 44 submission were received this year. On the other hand we feel that the average quality of submissions has gone up, and thus the program committee was able to select 18 good papers, only one less than the year before.

The program committee did not meet physically, and all discussion was done using a Web-driven data base system. Despite some mixed feelings there is an overall tendency to appreciate the extra time available for giving papers a second look and really going into comments made by other program committee members.

I want to thank my fellow program committee members for the work they have put into the refereeing process and the valuable feedback they have given to authors. I want to thank the referees for their work and many detailed comments, and finally I want to thank everyone who has submitted a paper: without authors, no conference.

Utrecht, January 1999

Doaitse Swierstra
ESOP'99 Chairman

Program Committee:

Ralph Back, TUCS, Turku, Finland
Roland Backhouse, Eindhoven University of Technology, The Netherlands
Franǫis Bourdoncle , Ecole des Mines de Paris, France
Luca Cardelli, Microsoft Research, Cambridge, UK
Andrew Gordon, Microsoft Research, Cambridge, UK
John Hughes, Chalmers University of Technology, Götenborg, Sweden
John Launchbury, Oregon Graduate Institute, Portland, OR, USA
Torben Mogensen, DIKU, Copenhagen, Denmark
Oege de Moor, Oxford, UK
Oscar Nierstrasz, University of Bern, Switzerland
José Oliveira, Un. Minho, Braga, Portugal
Maurizio Proietti, IASI-CNR, Rome, Italy
Gert Smolka, Universität des Saarlandes, Germany
Doaitse Swierstra (Chair), Utrecht University, The Netherlands

Referees for ESOP'99

Salvador Abreu
Franz Achermann
Hassan Aït-Kaci
Paulo Almeida
Luís Barbosa
Gilles Barthe
Richard Bird
Frank S. de Boer
Lex Bijlsma
Martin Buchi
Pierre Casteran
Nicoletta Cocco
Marco Comini
Patrick Cousot
Olivier Danvy
Thorsten Ehm
Conal Elliott
Andrzej Filinski
Sigbjorn Finne
Alexandre Frey
Larske Fredlund
Simon Gay
Georges Gonthier
Deepak Goyal
Kees Hemerik
Pedro Henriques
Stephen Jarvis
Neil Jones
Søren B. Lassen
Ranko Lazic
Xavier Leroy
Markus Lumpe
Armando Matos
Michel Mauny
Guy McCusker
Andy Moran
Carlos Moreno
Enrico Nardelli
Robb Nebbe
Martijn Oostdijk
Lars Pareto
Alberto Pettorossi
Corin Pitcher
Andreas Podelski
KVS Prasad
Wishnu Prasetya
Francesco Ranzato
António Ravara
Jon G. Riecke
Peter Sewell
Silvija Seres
Ganesh Sittampalam
Harald Søndergaard
Morten Heine Sørensen
Vasco Vasconcelos
Björn Victor
Mads Tofte
Wolfgang Weck
Joakim von Wright

Foreword

ETAPS'99 is the second 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 comprises five conferences (FOSSACS, FASE, ESOP, CC, TACAS), four satellite workshops (CMCS, AS, WAGA, CoFI), seven invited lectures, two invited tutorials, and six contributed tutorials.

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 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 programme committee and independent 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. As an experiment, ETAPS'99 also includes two invited tutorials on topics of special interest. 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 have hitherto been addressed in separate meetings.

ETAPS'99 has been organized by Jan Bergstra of CWI and the University of Amsterdam together with Frans Snijders of CWI. Overall planning for ETAPS'99 was the responsibility of the ETAPS Steering Committee, whose current membership is:

> André Arnold (Bordeaux), Egidio Astesiano (Genoa), Jan Bergstra (Amsterdam), Ed Brinksma (Enschede), Rance Cleaveland (Stony Brook), Pierpaolo Degano (Pisa), Hartmut Ehrig (Berlin), José Fiadeiro (Lisbon), Jean-Pierre Finance (Nancy), Marie-Claude Gaudel (Paris), Susanne Graf (Grenoble), Stefan Jähnichen (Berlin), Paul Klint (Amsterdam), Kai Koskimies (Tampere), Tom Maibaum (London), Ugo Montanari (Pisa), Hanne Riis Nielson (Aarhus), Fernando Orejas (Barcelona), Don Sannella (Edinburgh), Gert Smolka (Saarbrücken), Doaitse Swierstra (Utrecht), Wolfgang Thomas (Aachen), Jerzy Tiuryn (Warsaw), David Watt (Glasgow)

ETAPS'98 has received generous sponsorship from:

KPN Research
Philips Research
The EU programme "Training and Mobility of Researchers"
CWI
The University of Amsterdam
The European Association for Programming Languages and Systems
The European Association for Theoretical Computer Science

I would like to express my sincere gratitude to all of these people and organizations, the programme committee members of the ETAPS conferences, the organizers of the satellite events, the speakers themselves, and finally Springer-Verlag for agreeing to publish the ETAPS proceedings.

Edinburgh, January 1999

Donald Sannella
ETAPS Steering Committee Chairman

Table of Contents

Invited Paper

Functional Reactive Programming 1
Paul Hudak

Regular Contributions

A Decidable Logic for Describing Linked Data Structures 2
Michael Benedikt, Thomas Reps and Mooly Sagiv

Interprocedural Control Flow Analysis 20
Flemming Nielson and Hanne Riis Nielson

A Per Model of Secure Information Flow in Sequential Programs 40
A. Sabelfeld and D. Sands

Quotienting Share for Dependency Analysis 59
Andy King, Jan-Georg Smaus and Pat Hill

Types and Subtypes for Client-Server Interactions 74
Simon Gay and Malcolm Hole

Types for Safe Locking 91
Cormac Flanagan and Martín Abadi

Constructor Subtyping 109
Gilles Barthe and Maria João Frade

Safe and Principled Language Interoperation 128
Valery Trifonov and Zhong Shao

Deterministic Expressions in C 147
Michael Norrish

A Programming Logic for Sequential Java 162
Arnd Poetzsch-Heffter and Peter Müller

Set-Based Failure Analysis for Logic Programs and Concurrent Constraint Programs 177
Andreas Podelski, Witold Charatonik and Martin Müller

An Idealized MetaML: Simpler, and More Expressive 193
Eugenio Moggi, Walid Taha, Zine El-Abidine Benaissa and Tim Sheard

Type-Based Decompilation 208
Alan Mycroft

An Operational Investigation of the CPS Hierarchy 224
Olivier Danvy and Zhe Yang

Higher-Order Code Splicing 243
Peter Thiemann

Expressing Structural Properties as Language Constructs 258
Shriram Krishnamurthi, Yan-David Erlich and Matthias Felleisen

Polytypic Compact Printing and Parsing 273
Patrik Jansson and Johan Jeuring

Dynamic Programming via Static Incrementalization 288
Yanhong A. Liu and Scott D. Stoller

Author Index **307**

Functional Reactive Programming

Paul Hudak

Yale University, New Haven, CT 06518, USA
paul.hudak@yale.edu
www.cs.yale.edu/users/hudak.html

Abstract. *Functional reactive programming*, or FRP, is a style of programming based on two key ideas: continuous time-varying *behaviors*, and event-based *reactivity*. FRP is the essence of *Fran* [1,2], a domain-specific language for functional reactive graphics and animation, and has recently been used in the design of *Frob* [3,4], a domain-specific language for functional vision and robotics. In general, FRP can be viewed as an interesting language for describing *hybrid systems*, which are systems comprised of both analog (continuous) and digital (discrete) subsystems. Continuous behaviors can be thought of simply as functions from time to some value: `Behavior a = Time -> a`. For example: an image behavior may represent an animation; a Cartesian-point behavior may be a mouse; a velocity-vector behavior may be the control vector for a robot; and a tuple-of-distances behavior may be the input from a robot's sonar array. Both continuous behaviors and event-based reactivity have interesting properties worthy of independent study, but their integration is particularly interesting. At the core of the issue is that events are intended to cause discrete shifts in *declarative* behavior; i.e. not just shifts in the state of reactivity. Being declarative, the natural desire is for everything to be first-class and higher-order. But this causes interesting clashes in frames of reference, especially when time and space transformations are applied. In this talk the fundamental ideas behind FRP are presented, along with a discussion of various issues in its formal semantics.
This is joint work with Conal Elliot at Microsoft Research, and John Peterson at Yale.

References

1. Conal Elliott. Modeling interactive 3D and multimedia animation with an embedded language. In *Proceedings of the first conference on Domain-Specific Languages*, pages 285–296. USENIX, October 1997.
2. Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, pages 163–173, June 1997.
3. John Peterson, Paul Hudak, and Conal Elliott. Lambda in motion: Controlling robots with haskell. In *First International Workshop on Practical Aspects of Declarative Languages*. SIGPLAN, Jan 1999.
4. A. Reid, J. Peterson, G. Hager, and P. Hudak. Prototyping real-time vision systems: An experiment in DSL design. To appear Proc. Int. Conference on Software Engineering, May 1999.

S.D. Swierstra (Ed.): ESOP/ETAPS'99, LNCS 1576, pp. 1–1, 1999.
© Springer-Verlag Berlin Heidelberg 1999

A Decidable Logic for Describing Linked Data Structures

Michael Benedikt[1], Thomas Reps[2], and Mooly Sagiv[3]

[1] Bell Laboratories, Lucent Technologies, benedikt@research.bell-labs.com
[2] University of Wisconsin, reps@cs.wisc.edu
[3] Tel-Aviv University, sagiv@math.tau.ac.il.

Abstract. This paper aims to provide a better formalism for describing properties of linked data structures (e.g., lists, trees, graphs), as well as the intermediate states that arise when such structures are destructively updated. The paper defines a new logic that is suitable for these purposes (called L_r, for "logic of reachability expressions"). We show that L_r is decidable, and explain how L_r relates to two previously defined structure-description formalisms ("path matrices" and "static shape graphs") by showing how an arbitrary shape descriptor from each of these formalisms can be translated into an L_r formula.

1 Introduction

This paper aims to provide a better formalism for describing properties of linked data structures (e.g., lists, trees, graphs). In past work with the same motivation, a variety of different formalisms have been developed — including "static shape graphs" [14, 15, 17, 12, 3, 23, 1, 19, 27, 21, 20, 22], "path matrices" [9, 11], "graph types" [16], and the ADDS annotation formalism [10] — and several previously known formalisms have been exploited — including graph grammars [6] and monadic second-order logic [13]. For lack of a better term, we will use the phrase *structure-description formalisms* to refer to such formalisms in a generic sense.

In this paper, we define a new logic (called L_r, for "logic of reachability expressions"), and show that L_r is suitable for describing properties of linked data structures. We show that L_r is decidable. We also show in detail how L_r relates to two of the previously defined structure-description formalisms: In Section 3, we show how a generalization of Hendren's path-matrix descriptors [9, 11] can be represented by L_r formulae; in Section 4, we show how the variant of static shape graphs defined in [21] can be represented by L_r formulae. In this way, L_r provides insight into the expressive power of path matrices and static shape graphs.

The benefits of our work include the following:

- The logic L_r can be used as an annotation language to express loop invariants and pre- and post-conditions of statements and procedures. Annotations are important not only as a means of documenting programs, but also as the basis for analyzing and reasoning about programs in a modular fashion. Our work has two advantages:

S.D. Swierstra (Ed.): ESOP/ETAPS'99, LNCS 1576, pp. 2–19, 1999.
© Springer-Verlag Berlin Heidelberg 1999

  - The logic L_r is quite expressive (e.g., strictly more expressive than the formalism used by Hendren et al. [10]). The added expressibility is important for describing the intermediate states that arise when linked data structures are destructively updated.
  - The logic L_r is *decidable*, which means that there is an algorithm that determines, for every formula in the logic, if the formula is satisfiable. In other words, it is possible to determine if there is any store at all that satisfies a given formula. In principle, this ability can be used to provide some sanity checks on the formulae that a user employs — e.g., a warning can be issued if the user employs a formula that is unsatisfiable.
- Our work makes contributions on the question of *extracting information from the results of program analysis*. Although the subject of the paper is not primarily *algorithms for analyzing* programs that manipulate linked data structures, the decidability of L_r — together with the constructions given in Sections 3 and 4 for encoding other structure-description formalisms in L_r — has interesting consequences for extracting information from the results of program analyses: L_r provides a way to *amplify* the results obtained from known pointer-analysis, alias-analysis, and shape-analysis algorithms in the following ways:
  - For a structure-description formalism in which each structure descriptor corresponds to an L_r formula, as is the case for path matrices (Section 3) and static shape graphs (Section 4), it is possible to determine if there is any store at all that corresponds to a given structure descriptor. This lets us determine whether a given structure descriptor contains any useful information.
  - Pointer-analysis, alias-analysis, and shape-analysis algorithms necessarily compute structure descriptors that over-approximate the pointer/alias/shape relationships that actually arise. This kind of loss of precision is intrinsic to static-analysis; however, many of the techniques that have been proposed in the literature have the feature that *additional imprecision* crops up when information is extracted from the structure descriptor for a particular program point. For instance, with the three-valued logic used for shape analysis in [20, 22], a formula that queries for a specific piece of information sometimes evaluates to "unknown", even when, in all of the stores that the static shape graph represents, the formula evaluates to a definite true or false value.
    For a structure-description formalism in which each structure descriptor corresponds to an L_r formula, decidability gives us a mechanism for *reading out information obtained by existing algorithms, without any additional loss of precision*: If φ is the formula that represents the shape descriptor and ψ is the formula that represents the query, we are interested in whether $\varphi \implies \psi$ always holds (or, equivalently, whether $\neg(\varphi \implies \psi)$ is unsatisfiable). Thus, in principle, the machinery developed in this paper allows us to take the structure descriptors computed by existing techniques, and extract information from them that is more precise than that envisioned by the inventors of these formalisms.

– For many of the structure-description formalisms used in the literature, very little is known about basic decision problems associated with them. Mapping a structure-description formalism F into L_r can provide a way to analyze many basic decision problems of F.

  For instance, a decision problem of special interest for structure-description formalisms that are used in abstract interpretation is the inclusion problem (i.e., whether the set of stores that structure descriptor D_1 represents is a subset of the set of stores that D_2 represents). When the inclusion problem is decidable, it is possible to check (i) whether one structure descriptor subsumes another (and hence the second need not be retained), and (ii) whether a simpler structure descriptor is a conservative replacement of a larger one, which is useful in widening. Thus, the inclusion problem is important for reducing both the time and space used during abstract interpretation.

  For a structure-description formalism in which each structure descriptor corresponds to an L_r formula, the inclusion of structure descriptor D_1 (represented by formula φ_1) in D_2 (represented by φ_2) is a matter of testing whether $\varphi_1 \implies \varphi_2$ always holds (or, equivalently, whether $\neg(\varphi_1 \implies \varphi_2)$ is unsatisfiable).

To date, our concern has been with developing the tools for describing properties of linked data structures and obtaining a logic that is decidable. We have developed a decision procedure for L_r, although this procedure does not yield a practical algorithm. We have not yet investigated the complexity of the decision problem for L_r, nor looked for heuristic methods with acceptable performance in practice, but we plan to do so in future work.

Two programs that will be used to illustrate our work are shown in Figure 1. The remainder of the paper is organized into six sections: Section 2 presents the logic we use for describing properties of linked data structures. Section 3 shows how a generalization of Hendren's path-matrix descriptors [9, 11] can be represented by L_r formulae. Section 4 shows how a variant of static shape graphs can be represented by L_r formulae. Section 5 discusses the issue of using L_r formulae to extract information from the results of program analyses. Section 6 gives a sketch of the proof that L_r is decidable. Section 7 discusses related work.

2 A Language for Stating Properties of Linked Data Structures

Definition 21 *Let PVar be the (finite) set of pointer variables in a given program. Let Sel be the set of* **pointer selectors** *(i.e., pointer-valued fields of structures) used in the program. We define the* **alphabet** Σ *to be the following finite set of symbols:* $\Sigma = Sel \cup \{\texttt{pvar?} \mid \texttt{pvar} \in PVar\} \cup \{\neg\texttt{pvar?} \mid \texttt{pvar} \in PVar\}$*, with the intended meaning that* `pvar`*? denotes the cell pointed to by the pointer variable* `pvar`*, and* ¬`pvar`*? denote cells not pointed to by* `pvar`*. A* **formula** *in*

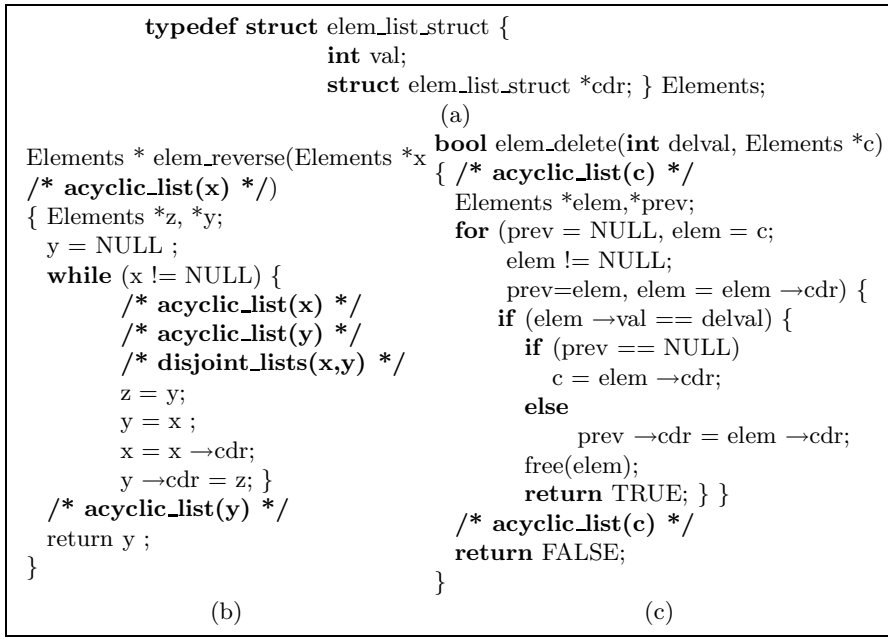


Fig. 1. (a) A C declaration of a linked-list type. (b)A program that uses destructive-updating operations to reverse a list. (c)A program that searches the list pointed to by variable `c` (using a "trailing pointer" `prev`) and deletes the first element whose `val` field equals `delval`.

the logic L_r is defined as follows:

$$
\begin{array}{rll}
\Phi ::= & \mathtt{pe}_1 = \mathtt{pe}_2 & \textit{equality of pointer exps.} \\
| & \mathtt{pe}_1\langle R\rangle\mathtt{pe}_2 & \textit{reachability constraint} \\
| & hs(\mathtt{pe}\langle R\rangle) & \textit{heap-sharing constraint} \\
| & al(\mathtt{pe}\langle R\rangle) & \textit{allocation constraint} \\
| & \neg\Phi & \textit{negation} \\
| & \Phi_1 \wedge \Phi_2 & \textit{conjunction} \\
| & \Phi_1 \vee \Phi_2 & \textit{disjunction} \\
| & \Phi_1 \implies \Phi_2 & \textit{implication}
\end{array}
\qquad
\begin{array}{rll}
R ::= & \epsilon & \textit{empty path} \\
| & \emptyset & \textit{empty lang.} \\
| & \sigma & \sigma \in \Sigma \\
| & R_1.R_2 & \textit{concat.} \\
| & R_1|R_2 & \textit{union} \\
| & R^* & \textit{Kleene star} \\
\mathtt{pe} ::= & \mathtt{pvar} & \textit{pointer var.} \\
| & \mathtt{pe.sel} & \mathtt{sel} \in \mathit{Sel}
\end{array}
$$

We call R terms **routing expressions**, *and refer to occurrences of* `pvar`? *and* ¬`pvar`? *in routing expressions as* **pointer-variable interrogations**.

We also use several shorthand notations: $hs(\mathtt{p})$ and $al(\mathtt{p})$ are shorthands for $hs(\mathtt{p}\langle\epsilon\rangle)$ and $al(\mathtt{p}\langle\epsilon\rangle)$, respectively. Similarly, $hs(\mathtt{p.sel})$ and $al(\mathtt{p.sel})$ are shorthands for $hs(\mathtt{p}\langle\mathtt{sel}\rangle)$ and $al(\mathtt{p}\langle\mathtt{sel}\rangle)$, respectively. $\mathtt{pe}_1 \neq \mathtt{pe}_2$ is a shorthand for $\neg(\mathtt{pe}_1 = \mathtt{pe}_2)$. $\Phi_1 \Leftrightarrow \Phi_2$ is a shorthand for $(\Phi_1 \implies \Phi_2) \wedge (\Phi_2 \implies \Phi_1)$.

Example 22 For a pointer variable `x`, the formula

$$acyclic\_list(\mathtt{x}) \stackrel{\text{def}}{=} \neg\mathtt{x}\langle\mathtt{cdr}^+\rangle\mathtt{x} \wedge \neg hs(\mathtt{x}\langle\mathtt{cdr}^*\rangle)$$

states that x points to an unshared acyclic list. The term $\neg \mathtt{x}\langle \mathtt{cdr}^+ \rangle \mathtt{x}$ signifies that a traversal that starts with the cell pointed to by x and follows one or more cdr fields never returns to the cell pointed to by x. The term $\neg hs(\mathtt{x}\langle \mathtt{cdr}^* \rangle)$ signifies that a traversal that starts with the cell pointed to by x and follows zero or more cdr fields never leads to a cell that is "heap shared". (A cell c is "heap shared" if two or more cells have fields that point to c, or if there is a cell c' such that $c'.\mathtt{car}$ and $c'.\mathtt{cdr}$ both point to c [15, 3, 21, 20].)

Thus, a loop invariant for program elem_reverse can be written as follows:

$$\begin{array}{ll} & ((al(\mathtt{y.cdr}) \vee al(\mathtt{z})) \implies \mathtt{y.cdr} = \mathtt{z}) \\ \wedge & \mathit{acyclic\_list}(\mathtt{x}) \wedge \mathit{acyclic\_list}(\mathtt{y}) \\ \wedge & \neg \mathtt{x}\langle \mathtt{cdr}^* \rangle \mathtt{y} \wedge \neg \mathtt{x}\langle \mathtt{cdr}^* \rangle \mathtt{z} \wedge \neg \mathtt{y}\langle \mathtt{cdr}^* \rangle \mathtt{x} \wedge \neg \mathtt{z}\langle \mathtt{cdr}^* \rangle \mathtt{x} \end{array} \tag{1}$$

The first line of (1) states that y.cdr and z refer to the same list element when either one is allocated. The subformulae on the last line of (1) states that the x-list is disjoint from both the y-list and the z-list.

Example 23 A loop invariant for program elem_delete can be written as follows:

$$\begin{array}{ll} & \mathit{acyclic\_list}(\mathtt{c}) \wedge \mathtt{c}\langle \mathtt{cdr}^* \rangle \mathtt{elem} \\ \wedge & al(\mathtt{prev}) \implies (\mathtt{c}\langle \mathtt{cdr}^* \rangle \mathtt{prev} \wedge \mathtt{prev.cdr} = \mathtt{elem}) \\ \wedge & \neg al(\mathtt{prev}) \iff \mathtt{elem} = \mathtt{c} \end{array} \tag{2}$$

The subformula $\mathtt{c}\langle \mathtt{cdr}^* \rangle \mathtt{elem}$ states that elem points somewhere in the list pointed to by c. The subformula on the last line of (2) states that prev is allocated (i.e., not NULL) if and only if elem and c point to different locations. From this, we can conclude that the location released by the statement free(elem) cannot be pointed to by c.

The use of pointer-variable interrogations in routing expressions will be illustrated in Examples 33 and 36.

We now define the semantics of L_r formulae:

Definition 24 *A store S can be represented by a tuple $\langle Loc^S, env^S, \iota^S \rangle$, where Loc^S is a set of locations, and env^S and ι^S are functions*

$$\begin{array}{rcl} env^S & : & PVar \to (Loc^S \cup \{\bot\}) \\ \iota^S & : & Sel \to (Loc^S \cup \{\bot\}) \to (Loc^S \cup \{\bot\}), \end{array}$$

where ι^S is strict in $\bot$.

The meaning of a pointer expression pe *in a given store S, denoted by $[\![\mathtt{pe}]\!]^S$ (where $[\![\mathtt{pe}]\!]^S \in (Loc^S \cup \{\bot\})$), is defined inductively, as follows:*

$$\begin{array}{c} [\![\mathtt{pvar}]\!]^S = env^S(\mathtt{pvar}) \\ [\![\mathtt{pe.sel}]\!]^S = \iota^S(\mathtt{sel})([\![\mathtt{pe}]\!]^S) \end{array}$$

The language $L(R)$ of a routing expression R is defined as is usual for regular expressions. However, because a word in $L(R)$ can contain occurrences of pointer-variable interrogations of the form pvar? *and* ¬pvar?*, the meaning of a word is*

slightly nonstandard: The meaning of a word in a given store S, denoted by $[\![w]\!]^S$ (where $[\![w]\!]^S$: $(Loc^S \cup \{\bot\}) \to (Loc^S \cup \{\bot\})$), is defined inductively, as follows:

$$
\begin{aligned}
[\![\epsilon]\!]^S(l) &= l \\
[\![w.\mathtt{sel}]\!]^S(l) &= \iota^S(\mathtt{sel})([\![w]\!]^S(l)) \\
[\![w.\mathtt{pvar?}]\!]^S(l) &= \begin{cases} [\![w]\!]^S(l) & \text{if } env(\mathtt{pvar}) = [\![w]\!]^S(l) \\ \bot & \text{otherwise} \end{cases} \\
[\![w.\neg\mathtt{pvar?}]\!]^S(l) &= \begin{cases} [\![w]\!]^S(l) & \text{if } env(\mathtt{pvar}) \neq [\![w]\!]^S(l) \\ \bot & \text{otherwise} \end{cases}
\end{aligned}
$$

The meaning of formula in a given store S is defined inductively, as follows:

$$
\begin{aligned}
[\![\mathtt{pe}_1 = \mathtt{pe}_2]\!]^S &= ([\![\mathtt{pe}_1]\!]^S = [\![\mathtt{pe}_2]\!]^S) \\
[\![\mathtt{pe}_1\langle R\rangle\mathtt{pe}_2]\!]^S &= \text{there exists } w \in L(R) \text{ s.t. } [\![w]\!]^S([\![\mathtt{pe}_1]\!]^S) = [\![\mathtt{pe}_2]\!]^S \\
&\quad \text{and } [\![\mathtt{pe}_2]\!]^S \in Loc \\
[\![hs(\mathtt{pe}\langle R\rangle)]\!]^S &= \text{there exists } w \in L(R) \text{ s.t. } [\![w]\!]^S([\![\mathtt{pe}]\!]^S) = l \text{ and } l \in Loc \text{ and} \\
&\quad \text{there exist } l_1, l_2 \in Loc, \mathtt{sel}_1, \mathtt{sel}_2 \in Sel \text{ s.t. } \iota^S(\mathtt{sel}_1)(l_1) = l \\
&\quad \text{and} \iota^S(\mathtt{sel}_2)(l_2) = l \text{ and either (i) } l_1 \neq l_2 \text{ or (ii) } \mathtt{sel}_1 \neq \mathtt{sel}_2 \\
[\![al(\mathtt{pe}\langle R\rangle)]\!]^S &= \text{there exists } w \in L(R) \text{ such that } [\![w]\!]^S([\![\mathtt{pe}]\!]^S) \in Loc \\
[\![\neg\Phi]\!]^S &= [\![\Phi]\!]^S \text{ is false} \\
[\![\Phi_1 \wedge \Phi_2]\!]^S &= [\![\Phi_1]\!]^S \text{ is true and } [\![\Phi_2]\!]^S \text{ is true} \\
[\![\Phi_1 \vee \Phi_2]\!]^S &= [\![\Phi_1]\!]^S \text{ is true or } [\![\Phi_2]\!]^S \text{ is true} \\
[\![\Phi_1 \implies \Phi_2]\!]^S &= [\![\neg\Phi_1]\!]^S \text{ is true or } [\![\Phi_2]\!]^S \text{ is true}
\end{aligned}
$$

3 Representing Path Matrices via Formulae

In this section, we study the relationship between the logic L_r and a variant of the *path-matrix* structure-description formalism [9, 11]. A path matrix records information about the (possibly empty) set of paths that exist between pairs of pointer variables in a program. The version of path matrices described below is a generalization of the original version described in [9, 11]. We show that every path matrix (of the extended version of the formalism) can be represented by a formula in logic L_r.

Definition 31 *A path matrix pm contains an entry $pm[\mathtt{x}, \mathtt{y}]$ for every pair of pointer-valued program variables, $\mathtt{x}$ and $\mathtt{y}$. An entry $pm[\mathtt{x}, \mathtt{y}]$ describes the set of paths from the cell pointed to by $\mathtt{x}$ to the cell pointed to by $\mathtt{y}$. An entry $pm[\mathtt{x}, \mathtt{y}]$ has a value of the form $\langle R, Q\rangle$, where R is a regular expression over Σ, and Q is either "P" (standing for "possible path") or "D" (standing for "definite" path).*

The notions of "possible paths" and "definite paths" are somewhat subtle (and the names "possible paths" and "definite paths", which we have adopted

from [9, 11], are somewhat misleading). In the discussion below, let S be a store that path matrix pm represents, and let $Paths_S(\mathtt{x}, \mathtt{y})$ denote the set of paths from the cell pointed to by program variable $\mathtt{x}$ to the cell pointed to by $\mathtt{y}$.

- An entry $pm[\mathtt{x}, \mathtt{y}]$ that has the value $\langle R_D, D\rangle$ means that there is a path p in S from the cell pointed to by program variable $\mathtt{x}$ to the cell pointed to by $\mathtt{y}$, such that $p \in L(R_D)$. In other words,

$$Paths_S(\mathtt{x}, \mathtt{y}) \cap L(R_D) \neq \emptyset. \tag{3}$$

  Note that only *one* of the paths in $L(R_D)$ need be a path in $Paths_S(\mathtt{x}, \mathtt{y})$ for $pm[\mathtt{x}, \mathtt{y}] = \langle R_D, D\rangle$ to be satisfied.
- An entry $pm[\mathtt{x}, \mathtt{y}]$ that has the value $\langle R_P, P\rangle$ means that $L(R_P)$ is an over-approximation to the set of paths from the cell pointed to by $\mathtt{x}$ to the cell pointed to by $\mathtt{y}$. In other words,

$$Paths_S(\mathtt{x}, \mathtt{y}) \subseteq L(R_P). \tag{4}$$

  An alternative way to think about this is as follows: What we really mean by "R_P represents possible paths in store S" is that $\overline{L(R_P)} = \Sigma^* - L(R_P)$ is a set of *impossible* paths of S: That is, an entry $pm[\mathtt{x}, \mathtt{y}]$ that has the value $\langle R_P, P\rangle$ means that none of the paths from the cell pointed to by $\mathtt{x}$ to the cell pointed to by $\mathtt{y}$ are in $\overline{L(R_P)}$. Thus, we have

$$Paths_S(\mathtt{x}, \mathtt{y}) \cap \overline{L(R_P)} = \emptyset. \tag{5}$$

  These two ways of looking at things are equivalent, as shown by the following derivation: $Paths_S(\mathtt{x}, \mathtt{y}) \cap \overline{L(R_P)} = \emptyset \implies Paths_S(\mathtt{x}, \mathtt{y}) - L(R_P) = \emptyset \implies Paths_S(\mathtt{x}, \mathtt{y}) \subseteq L(R_P)$.

It is instructive to consider some simple examples of possible path-matrix entries:

- An entry $pm[\mathtt{x}, \mathtt{y}]$ that has the value $\langle \emptyset, P\rangle$ represents the fact that there is no path in S from the cell pointed to by program variable $\mathtt{x}$ to the cell pointed to by $\mathtt{y}$.
- An entry $pm[\mathtt{x}, \mathtt{y}]$ that has the value $\langle \epsilon, D\rangle$ represents the fact that $\mathtt{x}$ and $\mathtt{y}$ are must-aliases, i.e., $\mathtt{x}$ and $\mathtt{y}$ must point to the same cell in all of the stores that the path matrix represents.
- In contrast, an entry $pm[\mathtt{x}, \mathtt{y}]$ with the value $\langle \epsilon, P\rangle$ represents the fact that $\mathtt{x}$ and $\mathtt{y}$ are may-aliases, i.e., $\mathtt{x}$ and $\mathtt{y}$ might point to the same cell in some of the stores that the path matrix represents, but it is also possible that in other stores that the path matrix represents, there is no path at all from the cell pointed to by $\mathtt{x}$ to the cell pointed to by $\mathtt{y}$.
- More generally, a value $\langle R, P\rangle$ for entry $pm[\mathtt{x}, \mathtt{y}]$, where $\epsilon \in L(R)$ means that $\mathtt{x}$ and $\mathtt{y}$ are may aliases. The language $L(R) - \{\epsilon\}$ represents other possible paths from the cell pointed to by $\mathtt{x}$ to the cell pointed to by $\mathtt{y}$, but it is also possible that in some of the stores that the path matrix represents, there is no path at all from the cell pointed to by $\mathtt{x}$ to the cell pointed to by $\mathtt{y}$.

Note that a path matrix represents a smaller set of stores if the language for a "D" entry is made smaller, and also if the language for a "P" entry is made smaller (see (3) and (4)).

Example 32 The following path matrix expresses a loop-invariant for the loop of `elem_reverse`:

pm	x	y	z
x	$\langle \epsilon, D \rangle$	$\langle \emptyset, P \rangle$	$\langle \emptyset, P \rangle$
y	$\langle \emptyset, P \rangle$	$\langle \epsilon, D \rangle$	$\langle \mathtt{cdr}, D \rangle$
z	$\langle \emptyset, P \rangle$	$\langle \emptyset, P \rangle$	$\langle \epsilon, D \rangle$

(6)

The fact that $pm[\mathtt{y}, \mathtt{z}]$ is $\langle \mathtt{cdr}, D \rangle$ signifies that $\mathtt{y} \rightarrow \mathtt{cdr}$ must point to the cell that `z` points to.

Example 33 The following path matrix expresses a loop-invariant for the loop of `elem_delete`:

pm	prev	elem	c
prev	$\langle \epsilon, D \rangle$	$\langle \mathtt{cdr}, P \rangle$	$\langle \emptyset, P \rangle$
elem	$\langle \emptyset, P \rangle$	$\langle \epsilon, D \rangle$	$\langle \emptyset, P \rangle$
c	$\langle \mathtt{cdr}^*, P \rangle$	$\langle \epsilon \vert \mathtt{cdr}^*.\mathtt{prev}?.\mathtt{cdr}, P \rangle$	$\langle \epsilon, D \rangle$

(7)

The fact that $pm[\mathtt{prev}, \mathtt{elem}]$ is $\langle \mathtt{cdr}, P \rangle$ signifies that $\mathtt{prev} \rightarrow \mathtt{cdr}$ may point to the cell pointed to by `elem`, but may also point to a cell that `elem` does not point to; in fact, the latter is the case at the beginning of the first loop iteration. Similarly, the fact that $pm[\mathtt{c}, \mathtt{prev}]$ is $\langle \mathtt{cdr}^*, P \rangle$ signifies that `prev` may be reachable from `c`. The fact that $pm[\mathtt{c}, \mathtt{elem}]$ entry is $\langle \epsilon | \mathtt{cdr}^*.\mathtt{prev}?.\mathtt{cdr}, P \rangle$ signifies that either `c` and `elem` point to the same cell, or else that as we traverse the list pointed to by `c`, we first reach a cell pointed to by `prev` and then the cell pointed to by `elem`.

Remark. The routing expressions that we allow in path matrices are more general than the ones allowed in [9, 11] in the following way:

- We allow arbitrary alternations and not just `car|cdr`.
- We follow [13] in allowing pointer-variable interrogations (e.g., `prev`, ¬`prev`) in routing expressions. This comes in handy in cases where several paths depend on each other (cf. the $pm[\mathtt{c}, \mathtt{elem}]$ entry in path matrix (7)).

(The use of a less-general language of routing expressions in [9, 11] was motivated by the need to be able to compute efficiently a safe approximation to the path matrix at every program point.)

Since path matrices are an intuitive notation, we will not spend the space in directly formalizing the meaning of path matrices in terms of sets of stores. Instead, we now define the meaning of a path matrix by a formula in our language that characterizes the set of stores that a path matrix represents.

Definition 34 *For a regular expression R, let $\overline{R}$ denote the complement of R, i.e., a regular expression such that $L(\overline{R}) = \overline{L(R)} = \Sigma^* - L(R)$. For a path matrix pm, we define the formula φ_{pm} as follows:*

$$\varphi_{pm} \stackrel{\text{def}}{=} \bigwedge_{\mathtt{x},\mathtt{y} \in PVar, \langle R, D\rangle \in pm[\mathtt{x},\mathtt{y}]} \mathtt{x}\langle R\rangle\mathtt{y} \wedge \bigwedge_{\mathtt{x},\mathtt{y} \in PVar, \langle R, P\rangle \in pm[\mathtt{x},\mathtt{y}]} \neg\mathtt{x}\langle \overline{R}\rangle\mathtt{y} \qquad (8)$$

This definition is justified by the discussion that follows Definition 31.

Example 35 Path matrix (6), which expresses a loop-invariant for the loop of `elem_reverse` (see Example 32), corresponds to the following formula:

$$\begin{array}{rcrcr} \mathtt{x}\langle\epsilon\rangle\mathtt{x} & \wedge & \neg\mathtt{x}\langle\Sigma^*\rangle\mathtt{y} & \wedge & \neg\mathtt{x}\langle\Sigma^*\rangle\mathtt{z} \\ \wedge\ \neg\mathtt{y}\langle\Sigma^*\rangle\mathtt{x} & \wedge & \mathtt{y}\langle\epsilon\rangle\mathtt{y} & \wedge & \mathtt{y}\langle\mathtt{cdr}\rangle\mathtt{z} \\ \wedge\ \neg\mathtt{z}\langle\Sigma^*\rangle\mathtt{x} & \wedge & \neg\mathtt{z}\langle\Sigma^*\rangle\mathtt{y} & \wedge & \mathtt{z}\langle\epsilon\rangle\mathtt{z} \end{array} \qquad (9)$$

Formula (9) is less informative than the loop-invariant given as Formula (1) of Example 22. For example, with Formula (9) it is not known that `x` points to a list, because cyclic stores of the form shown in Figure 2 also satisfy (9).

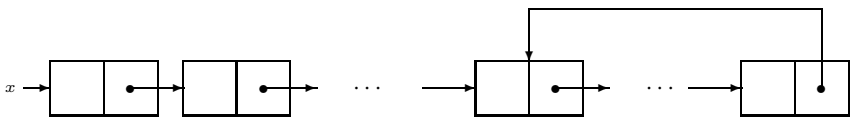


Fig. 2. A store with a shared node.

Example 36 Path matrix (10), which expresses a loop-invariant for the loop of `elem_delete` (see Example 33), corresponds to the following formula:

$$\begin{array}{rcrcrcr} \mathtt{prev}\langle\epsilon\rangle\mathtt{prev} & \wedge & \neg\mathtt{prev}\langle\overline{\mathtt{cdr}}\rangle\mathtt{elem} & \wedge & \neg\mathtt{prev}\langle\Sigma^*\rangle\mathtt{c} \\ \wedge\ \neg\mathtt{elem}\langle\Sigma^*\rangle\mathtt{prev} & \wedge & \mathtt{elem}\langle\epsilon\rangle\mathtt{elem} & \wedge & \mathtt{elem}\langle\Sigma^*\rangle\mathtt{c} \\ \wedge\ \neg\mathtt{c}\langle\overline{\mathtt{cdr}^*}\rangle\mathtt{prev} & \wedge & \neg\mathtt{c}\langle\overline{\epsilon|\mathtt{cdr}^*.\mathtt{prev}?.\mathtt{cdr}}\rangle\mathtt{elem} & \wedge & \mathtt{c}\langle\epsilon\rangle\mathtt{c} \end{array} \qquad (10)$$

Formula (10) is less informative than the loop-invariant given as Formula (2) of Example 23. In contrast to Formula (2), Formula (10) cannot be used to conclude that the use of `free` in `elem_delete` is correct; i.e., we cannot conclude that the location released by the statement `free(elem)` cannot be pointed to by `c`.

4 Representing Shape Graphs via Formulae

In this section, we study a structure-description formalism called *static shape graphs*, which, in addition to reachability information, allow certain "topological" properties of stores to be represented. There are many ways to define static shape graphs. For simplicity, we only consider the variant of static shape graphs defined in [21]. In Section 4.1, we give a formal definition of static shape graphs. Then, in Section 4.2, we construct a formula in L_r that exactly characterizes the set of stores represented by a static shape graph.

4.1 Static Shape Graphs

Below, we formally define static shape graphs. Unlike the stores defined in Section 2, static shape graphs are of an *a priori* bounded size, i.e., the number of shape nodes depends only of the size of the program being analyzed. This is needed by shape-analysis algorithms so that an iterative shape-analysis algorithm that computes static shape graphs for each program point will terminate.

Definition 41 *A* **static-shape-graph** *(SSG) is a finite directed graph that consists of two kinds of nodes —* **variables** *(i.e., PVar) and* **shape-nodes** *— and two kinds of edges —* **variable-edges** *and* **selector-edges**. *A shape-graph is represented by a quadruple* $\langle shapeNodes, E_v, E_s, is \rangle$, *where:*

- *shapeNodes is a finite set of shape nodes. Every shape node* $n \in ShapeNodes$ *has the form* $n = n_X$ *where* $X \subseteq PVar$. *Such a node describes the cells that are simultaneously pointed to by all the pointer variables in* X.
  Graphically, we denote shape nodes by circles. The node $n_\emptyset$ *is the "summary-node" since it represents all the cells that are not directly pointed to by any pointer variable, and therefore it is represented by a dotted circle.*
- E_v *is the graph's set of variable-edges, each of which is denoted by a pair of the form* $[\mathtt{x}, n_X]$, *where* $\mathtt{x} \in PVar$ *and* $n_X \in shapeNodes$. *We assume that for every* $\mathtt{x} \in PVar$, *at most one variable-edge* $[\mathtt{x}, n_X] \in E_v$ *exists and* $\mathtt{x} \in X$.
  Graphically, we denote variable-edges by solid edges since they must exist.
- E_s *is the graph's set of selector-edges, each of which is denoted by a triple of the form* $\langle n_X, sel, n_Y \rangle$, *where* $n_X, n_Y \in shapeNodes$ *and* $sel \in \{car, cdr\}$. *We assume that for every* $\mathtt{x} \in PVar$, $sel \in \{car, cdr\}$, *and shape node* n_X *such that* $[\mathtt{x}, n_X] \in E_v$, *at most one selector-edge,* $\langle n_X, sel, n_Y \rangle \in E_s$ *exists. In contrast, there may be many selector-edges* $\langle n_\emptyset, sel, n_Y \rangle \in E_s$ *corresponding to different selector-edges emanating from cells represented by* $n_\emptyset$.
  Graphically, we denote selector-edges by dotted edges since they may or may not exist.
- *is (standing for "is shared") is a function of type* $shapeNodes \to \{false, true\}$. *It serves as a constraint to restrict the set of stores represented by a shape graph. When* $n_\emptyset$ *has more than one incoming selector edge and yet* $is(n_\emptyset) = false$, *we know that, for any memory cell* c *represented by* $n_\emptyset$, *at most one of the concrete representatives of these selector-edges can be an incoming edge of* c.

Graphically, we denote the fact that n_X is a shared node by putting "$is(n_X)$" inside the circle.

Example 42 The SSG that represents the store shown in Figure 2 is shown in Figure 3.

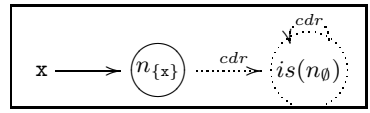


Fig. 3. The SSG that corresponds to the store shown in Figure 2.

4.2 From a Static Shape Graph to an L_rFormula

We are now ready to show how to construct the formula that captures the meaning of a static shape graph.

Definition 43 *Let $SG = \langle shapeNodes, E_v, E_s, is\rangle$ be an SSG. We define the graph $\widehat{SG}$ to be a directed graph, $\widehat{SG} = (N, A, l)$, with edges labeled by letters in Σ, where:*

- *N contains two nodes $u.in$ and $u.out$ for every shape node u in $shapeNodes$.*
- *$A \subseteq N \times N$ contains the following two types of labeled edges:*
  - *For every shape node n_X such that $[p_1, n_X], [p_2, n_X], \cdots, [p_n, n_X] \in E_v$ and $[p_{n+1}, n_X], [p_{n+2}, n_X], \cdots, [p_{n+k}, n_X] \notin E_v$, there is an edge $\langle n_X.in, n_Y.out\rangle$, labeled by $(p_1?.p_2?.\cdots.p_n?.\neg p_{n+1}?.\neg p_{n+2}?\cdots.\neg p_{n+k}?)$.*
  - *If there is a selector-edge $\langle n_X, sel, n_Y\rangle \in E_s$, there is an edge $a = (n_X.out, n_Y.in)$ from $n_X.out$ into $N_Y.in$, labeled sel.*
- *l: $A \to \Sigma$ maps edges into their labels.*

For any two nodes $n_X, n_Y \in shapeNodes$, let $r_{n_X.in \to n_Y.out}$ be the regular path expression over Σ that describes paths in $\widehat{SG}$ from $n_X.in$ into $n_Y.out$ (which can be computed by well-known methods, e.g., [25, 24]). For a finite set of regular expressions $S = \{r_1, r_2, \ldots, r_n\}$, $Rsum_S$ denotes the regular expression $r_1|r_2|\cdots|r_n$. Finally, for a regular expression r, $\overline{r}$ is the regular expression over Σ that describes the non-existing words in r, i.e., $L(\overline{r}) = \Sigma^* - L(r)$. Let us define the following formulae to characterize the different aspects of SG

$$\Phi^+_{val} = \bigwedge_{\mathtt{x}\in PVar, [\mathtt{x},n_X]\in E_v} al(\mathtt{x}) \qquad \Phi^-_{val} = \bigwedge_{\mathtt{x}\in PVar, [\mathtt{x},n_X]\notin E_v} \neg al(\mathtt{x})$$
$$\Phi^+_{veq} = \bigwedge_{\mathtt{x},\mathtt{y}\in PVar, [\mathtt{x},n_X],[\mathtt{y},n_X]\in E_v} \mathtt{x}=\mathtt{y} \quad \Phi^-_{veq} = \bigwedge_{\mathtt{x},\mathtt{y}\in PVar, [\mathtt{x},n_X]\in E_v, [\mathtt{y},n_X]\notin E_v} \mathtt{x}\neq\mathtt{y}$$
$$\Phi^-_{pal} = \bigwedge_{\mathtt{x}\in PVar, [\mathtt{x},n_X]\in E_v} \neg al(\mathtt{x}\langle \overline{Rsum_{n_Y\in shapeNodes} r_{n_X.in\to n_Y.out}}\rangle)$$
$$\Phi^-_{r} = \bigwedge_{[\mathtt{x},n_X],[\mathtt{y},n_Y]\in E_v} \neg \mathtt{x}\langle \overline{r_{n_X.in\to n_Y.out}}\rangle \mathtt{y}$$
$$\Phi^-_{hs} = \bigwedge_{\mathtt{x}\in PVar, [\mathtt{x},n_X]\in E_v} \neg hs(\mathtt{x}\langle \overline{Rsum_{n_Y\in shapeNodes, is(v)=1} r_{n_X.in\to n_Y.out}}\rangle)$$

Finally, the formula $\Phi[SG]$ is the conjunction of these formulae.

Lemma 44 *For every store S and SSG SG, S is represented by SG if and only if $[\![\Phi[SG]]\!]^S$ is true.*

5 Extracting Information from Program Analyses via L_r Formulae

Many interesting properties of linked data structures can be expressed as L_r formulae:

- For example, the formula $\Psi \stackrel{\text{def}}{=} (x = x.cdr)$, expresses the property "`x` points to a cell that has a self-cycle". This information can be used by an optimizing compiler to determine whether it is profitable to generate a pre-fetch for the next element [18].
- It is possible to express in L_r that two pointer-access paths point to different memory cells (i.e., they are not may-aliases), which is important both for optimization and in tools for aiding software understanding.
- The reachability and sharing predicates can also be useful, for example, to improve the performance of garbage-collection algorithms and to parallelize programs.

In principle, L_r provides a uniform basis for using the results of analyses that yield either path matrices or static shape graphs in program optimizers and in tools for aiding software understanding. For instance, Figure 4 shows one of the SSGs SG that arises at the loop header in an analysis of `elem_reverse`. It can be shown that Ψ is not satisfiable by any store that is represented by SG. This means that `x` does not point to a cell that has a self-cycle in any of the stores that SG represents. This can be determined automatically with our approach by showing that $\Phi[SG] \wedge \Psi$ is not satisfiable. Similarly, by translating a path matrix M (obtained from a path-matrix-based program-analysis algorithm) into the corresponding L_r formula $\Phi[M]$ and checking whether $\Phi[M] \wedge \Psi$ is satisfiable, one can verify automatically whether `x` could point to a cell that has a self-cycle in any of the stores represented by M.

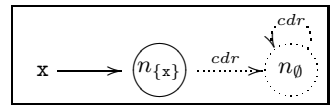


Fig. 4. An SSG, SG, that represents acyclic lists of length two or more that are pointed to by variable `x`.

6 The Decidability of L_r

Theorem 61 *L_r is decidable.*
Sketch of Proof: Prior to directly approaching the question of decidability of L_r, one first proves a normalization lemma showing that the routing expressions mentioned in formulae can be rewritten in such a way that they deal only with paths that avoid all nodes pointed by pointer expressions that are mentioned in the formula (i.e. pointer expressions that occur in some constraint or program variables that occur in some pointer-variable interrogation). That is, they assert only reachability of shared nodes or pointer expressions via paths that traverse nodes in the heap. One proves this normalization lemma by breaking down path expressions that may cross mentioned pointer expressions into component path expressions that do not cross mentioned pointer expressions.

The decidability of logic L_r follows from showing that that L_r has the *bounded model property*: that is, there is a computable numerical function f such that any sentence ϕ of L_r that is consistent has a model of size bounded by $f(|\phi|)$. This technique is one of the most common in logical decision procedures [2]. It immediately shows the existence of a crude decision procedure: one enumerates all possible stores of size $f(|\phi|)$ searching for a model. (Note that the approach sketched here is intended only to give a comprehensible demonstration of decidability, not to give a practical decision procedure.) The proof of the bounded model property proceeds by starting with an arbitrary concrete store G satisfying a formula ϕ and showing that G can be diminished to a model of size $|f(|\phi|)$ (for a particular f given in the proof) while preserving all atomic formulae in ϕ.

The normalization theorem above implies that in this shrinking process one only has to preserve properties that deal with paths through the heap (reachability, heap-sharing, etc.) and equalities and inequalities between a fixed set of pointer expressions. This shrinking is then done in three phases: first, the original store G is "pruned" to get a model that is a union of trees: in the process, some information about the sharing of nodes is lost, but extra labels are added to the nodes to maintain this information. These "auxiliary labels" indicate that certain nodes in the tree correspond to nodes associated with a particular pointer expression in the original store, and that certain nodes in the tree were shared in the original store.

We then make use of classical decidability results on reachability expressions on finite trees ([26], summarized also in [2]) to shrink each of these trees to smaller trees that satisfy the same properties as the union of trees produced in stage one. The "properties" mentioned here are obtained by taking the original reachability, heap-sharing, and allocation constraints and transforming them to expressions in monadic second-order logic that express how to reach the auxiliary labels mentioned above.

Finally, the shrunken set of trees are glued together to restore sharing information lost in the first phase: multiple nodes that have been annotated as associated with the same pointer expression are identified, and nodes that were annotated as being shared heap nodes are made into shared nodes. The normalization results are used in a crucial way in this glueing stage, since the glueing

can create many new paths within the represented store. Glueing cannot, however, create new paths through the heap in any store, since the glueing process only identifies nodes associated with pointer expressions mentioned in the formula (or in unshared paths leading to such nodes). Since normalization implies that we are only concerned with preserving the existence and nonexistence of paths that lie strictly within the heap, this is sufficient.

Figures 5 and 6 show how the proof might work for the formula Φ:

$$\Phi \stackrel{\text{def}}{=} x\langle \texttt{car}^* \rangle y \wedge x\langle (\texttt{cdr.cdr})^* \rangle y \\ \wedge \neg x\langle (\texttt{cdr.cdr.cdr})^* \rangle y \wedge y\langle \texttt{car}^* \rangle z \wedge y\langle (\texttt{cdr.cdr.cdr})^* \rangle z.$$

We start with a store in Figure 5 that satisfies Φ, and then prune it into a set of trees. The auxiliary labels y' and y'' keep track of the fact that these nodes in the tree must at some point be pointed to by y. In Figure 6, the trees are decreased in size, while preserving analogs of the reachability statements: e.g., the node labeled y can reach a copy of the node z with a $(\texttt{cdr.cdr.cdr})^*$ path, and x cannot reach a copy of y with a $(\texttt{cdr.cdr.cdr})^*$ path. In the final stage, the tree-like model is glued together to form a traditional store that satisfies Φ.

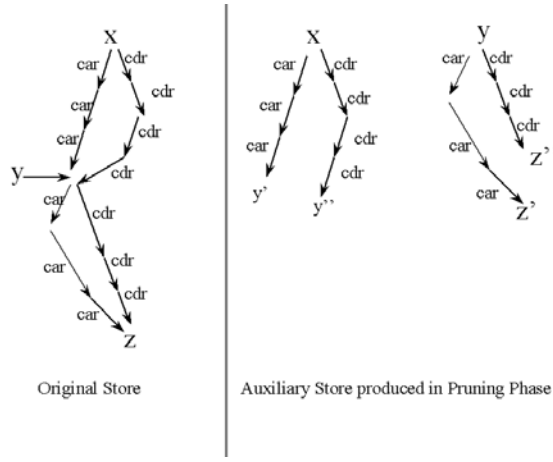


Fig. 5. The pruning stage of the proof

7 Related Work

Jensen et al. have also defined a decidable logic for describing properties of linked data structures [13]. It is interesting to compare the two approaches:

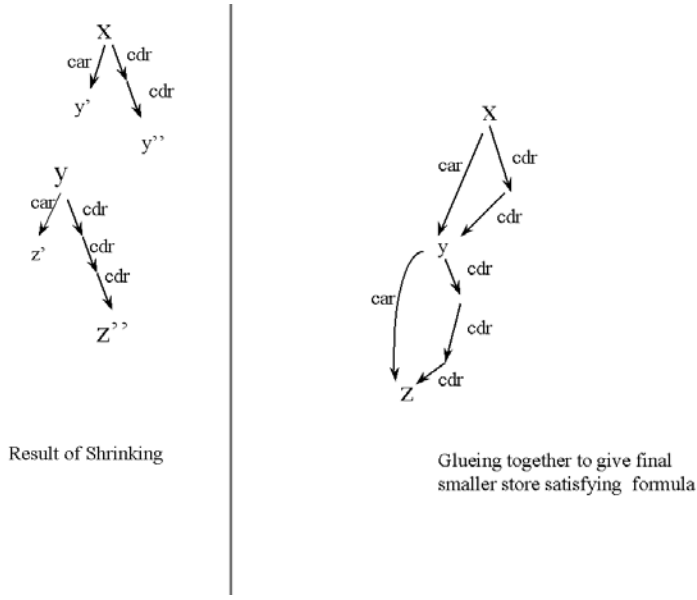


Fig. 6. The shrinking and glueing stages of the proof

- The logic of Jensen et al. allows quantifications on pointer expressions, which is forbidden in L_r. Instead, L_r allows stating both sharing constraints and allocation constraints. However, both of these can be encoded using their logic:
  - Sharing constraints that can be encoded using quantifications.
  - Allocation constraints can be encoded using tests for `NULL` in routing expressions.
- L_r imposes a limitation on routing expressions by forbidding testing for `NULL` and for garbage cells.
- On the other hand, L_r generalizes the logic of Jensen et al. in the following ways:
  - L_r allows multiple selectors, which enables L_r formulae to describe properties of general directed graphs as opposed to just lists.[1]
  - The reachability constraints in L_r formulae allow one to test simultaneous pointer inequities, which is crucial for capturing the strength of the variant of static shape graphs defined in [21].

In summary, the formulae of Jensen et al. are more expressive than L_r formulae, but they can only state properties of lists and trees, whereas L_r can state properties of arbitrary graph data structures.

Klarlund and Schwartzbach defined a language for defining *graph types*, which are tree data structures with non-tree links defined by auxiliary tree-path expressions [16]. In the application they envision, a programmer would be able

[1] [13] sketches an extension of their technique to trees, which involves multiple selectors, but they do not handle general directed graphs.

to declare variables of a given graph type, and write code to mutate the "tree backbone" of these structures. After a mutation operation, the runtime system would automatically apply an update operation that they define, which updates the non-tree links. The graph-type definition language is unsuitable for describing arbitrary store graphs, and the fact that the update operations are limited does not allow the programmer to write arbitrary pieces of code. (However, the latter property is a significant advantage for the intended application — a programming language supporting controlled destructive updating of graph data structures.)

The ADDS formalism of Hendren et al. is an annotation language for expressing loop invariants and pre- and post-conditions of statements and procedures [10]. From a programmer's point of view, an advantage of a logic like L_r over ADDS is that L_r is strong enough to allow stating properties of the kind that arise at intermediate points of a procedure, when a data structure is in the process of being traversed or destructively updated. For example, ADDS cannot be used to state the loop invariant of Example 22 because the relationship between `x` and `y` cannot be expressed. Because it is lacking in expressive power, ADDS is mainly useful as a documentation notation for type definitions, function arguments, and function return values. Hendren et al. propose to handle this limitation of ADDS by extending it with the ability to use a certain limited class of reachability properties between variables (of the kind used in the path matrices defined in [9]).

L_r goes beyond ADDS in the following ways:

- L_r permits stating properties of cyclic data structures.
- The routing expressions used in L_r formulae are general regular expressions (with pointer-variable interrogations).
- L_r is closed under both conjunction and negation. In contrast, ADDS cannot express the loop invariant in Example 23 because of the implication.

It should be noted that currently the notion of heap sharing in L_r is weaker than the ADDS notion of "dimension". It is easy to generalize L_r to include this concept without affecting its decidability. We did not do so in this paper because we wanted to stay with two selectors.

Finally, it should be noted that both L_r and ADDS do not allow stating connectivity properties of the form $\mathtt{x}\langle R_1\rangle = \mathtt{y}\langle R_2\rangle$. We believe that L_r can be generalized to handle this. (A limited form of such connectivity properties, restricted to be in the form $\mathtt{x}\langle(\mathtt{car}|\mathtt{cdr})^*\rangle = \mathtt{y}\langle(\mathtt{car}|\mathtt{cdr})^*\rangle$, was proposed in [8, 7].)

L_r is incomparable to Deutsch's symbolic aliases [4, 5]: Symbolic aliases allow the use of full-blown arithmetic, which cannot be used in a decidable logic. On the other hand, symbolic-alias expressions are not closed under negation. For instance, there is no way to express must-alias relationships using symbolic aliases. Thus, the loop invariant used in Example 23 cannot be expressed with symbolic aliases.

In [6], Fradet and Le Métayer use graph grammars to express interesting properties of the data structures of a C-like language. Graph grammars can be a

more natural formalism than logic for describing certain topological properties of stores. However, graph grammars are not closed under intersection and negation, and problems such as the inclusion problem are not decidable. In terms of expressive power, the structure-description formalism of [6] is incomparable to the one proposed in the present paper.

It should be noted that the approach given here is limited in several ways: The approach we have taken is to develop decidable, logic-based languages for capturing topological properties of a broad class of linked data structures. Undecidability results in predicate logic give many hard limitations on the expressiveness of such languages: For example, no such language exists that is closed under first-order quantification and boolean connectives. Although logic-based formalisms can be more succinct in expressing properties of linked data structures, they can also be more verbose; in particular, the output from our translation algorithms can be significantly more verbose than the input. For example, with the translation from a static shape graph SG into L_r formula $\Phi[SG]$ given in Section 4.2, the size of $\Phi[SG]$ can be exponential in $|SG|$.

There are a few properties that cannot be expressed in L_r, including: (i) whether a store contains a garbage cell (i.e., a cell not accessible from any variable), and (ii) whether a tree is balanced (or almost balanced, such as the condition used in AVL trees). It may be difficult to extend L_r to handle these sorts of properties. However, such properties go well beyond the scope of current optimizing compilers and tools for aiding software understanding.

Acknowledgements

This work was supported in part by the NSF under grants CCR-9625667 and CCR-9619219, by the United States-Israel Binational Science Foundation under grant 96-00337, and by a Vilas Associate Award from the Univ. of Wisconsin.

References

1. U. Assmann and M. Weinhardt. Interprocedural heap analysis for parallelizing imperative programs. In W. K. Giloi, S. Jähnichen, and B. D. Shriver, editors, *Programming Models For Massively Parallel Computers*, pages 74–82, Washington, DC, September 1993. IEEE Press.
2. E. Boerger, E. Graedel, and Y. Gurevich. *The Classical Decision Problem.* Springer-Verlag, 1996.
3. D.R. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 296–310, New York, NY, 1990. ACM Press.
4. A. Deutsch. A storeless model for aliasing and its abstractions using finite representations of right-regular equivalence relations. In *IEEE International Conference on Computer Languages*, pages 2–13, Washington, DC, 1992. IEEE Press.
5. A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 230–241, New York, NY, 1994. ACM Press.
6. Pascal Fradet and Daniel Le Metayer. Shape types. In *Symp. on Princ. of Prog. Lang.*, New York, NY, 1997. ACM Press.
7. R. Ghiya and L. Hendren. Putting pointer analysis to work. In *Symp. on Princ. of Prog. Lang.*, New York, NY, 1998. ACM Press.
8. R. Ghiya and L.J. Hendren. Connection analysis: A practical interprocedural heap analysis for C. In *Proc. of the 8th Int. Workshop on Lang. and Comp. for Par.*

Comp., number 1033 in Lec. Notes in Comp. Sci., pages 515–534, Columbus, Ohio, August 1995. Springer-Verlag.
9. L. Hendren. *Parallelizing Programs with Recursive Data Structures*. PhD thesis, Cornell Univ., Ithaca, NY, Jan 1990.
10. L. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and the transformation of imperative programs. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 249–260, New York, NY, June 1992. ACM Press.
11. L. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Trans. on Par. and Dist. Syst.*, 1(1):35–47, January 1990.
12. S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 28–40, New York, NY, 1989. ACM Press.
13. J.L. Jensen, M.E. Joergensen, N.Klarlund, and M.I. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, 1997.
14. N.D. Jones and S.S. Muchnick. Flow analysis and optimization of Lisp-like structures. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 4, pages 102–131. Prentice-Hall, Englewood Cliffs, NJ, 1981.
15. N.D. Jones and S.S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Symp. on Princ. of Prog. Lang.*, pages 66–74, New York, NY, 1982. ACM Press.
16. N. Klarlund and M. Schwartzbach. Graph types. In *Symp. on Princ. of Prog. Lang.*, New York, NY, January 1993. ACM Press.
17. J.R. Larus and P.N. Hilfinger. Detecting conflicts between structure accesses. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 21–34, New York, NY, 1988. ACM Press.
18. C.-K. Luk and T.C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, October 1996.
19. J. Plevyak, A.A. Chien, and V. Karamcheti. Analysis of dynamic structures for efficient parallel execution. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, volume 768 of *Lec. Notes in Comp. Sci.*, pages 37–57, Portland, OR, August 1993. Springer-Verlag.
20. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. Tech. Rep. TR-1383, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI, July 1998. Available at "http://www.cs.wisc.edu/wpis/papers/parametric.ps".
21. M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *Trans. on Prog. Lang. and Syst.*, 20(1):1–50, January 1998.
22. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Symp. on Princ. of Prog. Lang.*, 1999. Available at "http://www.cs.wisc.edu/wpis/papers/popl99.ps".
23. J. Stransky. A lattice for abstract interpretation of dynamic (Lisp-like) structures. *Inf. and Comp.*, 101(1):70–102, Nov. 1992.
24. R.E. Tarjan. Fast algorithms for solving path problems. *J. ACM*, 28(3):594–614, 1981.
25. R.E. Tarjan. A unified approach to path problems. *J. ACM*, 28(3):577–593, 1981.
26. J.W. Thatcher and J.B. Wright. Generalized finite automata with an application to a decision problem of second order logic. *Math. Syst. Theory*, 2:57–82, 1968.
27. E. Y.-B. Wang. *Analysis of Recursive Types in an Imperative Language*. PhD thesis, Univ. of Calif., Berkeley, CA, 1994.

Interprocedural Control Flow Analysis

Flemming Nielson and Hanne Riis Nielson

Department of Computer Science, University of Aarhus
Ny Munkegade, DK-8000 Aarhus C, Denmark
{fn,hrn}@daimi.au.dk
Web address: http://www.daimi.au.dk/~{fn,hrn}

Abstract. Control Flow Analysis is a widely used approach for analysing functional and object oriented programs. Once the applications become more demanding also the analysis needs to be more precise in its ability to deal with mutable state (or side-effects) and to perform polyvariant (or context-sensitive) analysis. Several insights in Data Flow Analysis and Abstract Interpretation show how to do so for imperative programs but the techniques have not had much impact on Control Flow Analysis. We show how to incorporate a number of key insights from Data Flow Analysis (involving such advanced interprocedural techniques as call strings and assumption sets) into Control Flow Analysis (using Abstract Interpretation to induce the analyses from a collecting semantics).

1 Introduction

Control Flow Analysis. The primary aim of Control Flow Analysis is to determine the set of functions that can be called at each application (e.g. `x e` where `x` is a formal parameter to some function) and has been studied quite extensively ([24,11,16] to cite just a few). In terms of paths through the program, one tries to *avoid* working with a complete flow graph where all call sites are linked to all function entries and where all function exits are linked to all return sites. Often this is accomplished by means of contours [25] (à la call strings [23] or tokens [12]) so as to improve the precision of the information obtained. One way to specify the analysis is to show how to generate a set of constraints [8,9,18,19] whose least solution is then computed using graph-based ideas. However, the majority of papers on Control Flow Analysis (e.g. [24,25,11,16]) do not consider side-effects — a notable exception being [10].

Data Flow Analysis. The *intra*procedural fragment of Data Flow Analysis ignores procedure calls and usually formulates a number of data flow equations whose least solution is desired (or sometimes the greatest when a dual ordering is used) [7]. It follows from Tarski's theorem [26] that the equations could equally well be presented as constraints: the least solution is the same.

The *inter*procedural fragment of Data Flow Analysis takes procedure calls into account and aims at treating calls and returns more precisely than mere goto's: if

S.D. Swierstra (Ed.): ESOP/ETAPS'99, LNCS 1576, pp. 20–39, 1999.
© Springer-Verlag Berlin Heidelberg 1999

a call site gives rise to analysing a procedure with a certain piece of information, then the resulting piece of information holding at the procedure exit should ideally only be propagated back to the return site corresponding to the actual call site (see Figure 1).

In other words, the *intra*procedural view is that all paths through a program are valid (and this set of paths is a regular language), whereas the *inter*procedural view is that only those paths will be valid where procedure entries and exits match in the manner of parentheses (and this set of paths is a proper context free language). Most papers on Data Flow Analysis (e.g. [23,13]) do not consider first-class procedures and therefore have no need for a component akin to Control Flow Analysis — a notable exception to this is [20].

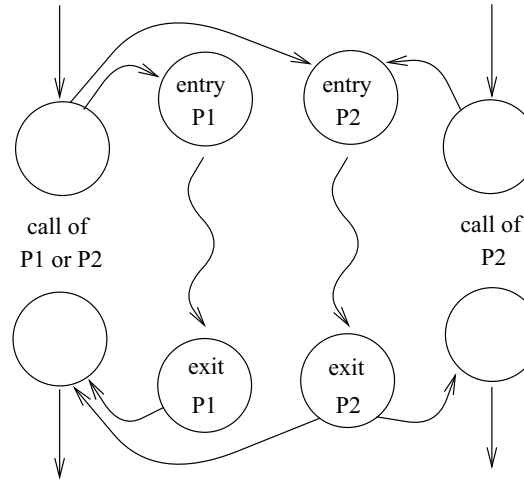


Fig. 1. Function call.

One approach deals with the interprocedural analysis by obtaining transfer functions for entire call statements [23,13] (and to some extent [3]). Alternatively, and as we shall do in this paper, one may dispense with formulating equations (or constraints) as the function level and extend the space of properties to include explicit context information.

- A widely used approach modifies the space of properties to include information about the pending procedure calls so as to allow the correct propagation of information at procedure exits even when taking a mainly intraprocedural approach; this is often formulated by means of call strings [23,27].
- A somewhat orthogonal approach modifies the space of properties to include information that is dependent on the information that was valid at the last procedure entry [20,14,21]; an example is the use of so-called assumption sets that give information about the actual parameters.

Abstract Interpretation. In Abstract Interpretation [4], the systematic development of program analyses is likely to span a spectrum from *abstract* specifications (like [16] in the case of Control Flow Analysis), over *syntax-directed* specifications

(as in the present paper), to actual *implementations* in the form of constraints being generated and subsequently solved (as in [8,9,18,19,6]). The main advantage of this approach is that semantic issues can be ignored in later stages once they have been dealt with in earlier stages. The first stage, often called the collecting semantics, is intended to cover a superset of the semantic considerations that are deemed of potential relevance for the analysis at hand. The purpose of each subsequent stage is to incorporate additional implementation oriented detail so as to obtain an analysis that satisfies the given demands on efficiency with respect to the use of time and space.

Aims. This paper presents an approach to program analysis that allows the simultaneous formulation of techniques for Control and Data Flow Analysis while taking the overall path recommended by Abstract Interpretation. To keep the specification compact we present the Control Flow Analysis in the form of a *succinct flow logic* [17]. Throughout the development we maintain a clear separation between *environment*-like data and *store*-like data so that the analysis more clearly corresponds to the semantics. As in [10] we add components for tracking the side-effects occurring in the program and for explicitly propagating environments; for the side-effects this gives rise to a *flow-sensitive* analysis and for the environments we might coin the term *scope-sensitive*.

The analysis makes use of mementoes (for expressing context information in the manner of [5]) that are general enough that both call string based approaches (e.g. [23,25]) and dependent data approaches (in the manner of assumption-sets [20,14]) can be obtained by merely approximating the space of mementoes; this gives rise to a *context-sensitive* analysis. The mementoes themselves are approximated using a surjective function and this approach facilitates describing the approximations between the various solution spaces using Galois connections as studied in the framework of Abstract Interpretation [3,4,1].

Overview. Section 2 presents the syntax of a functional language with side-effects. Section 3 specifies the abstract domains and Section 4 the analysis itself. In Section 5 we then show how the classical developments mentioned above can be obtained as Abstract Interpretations. Finally, Section 6 concludes. — The full version of this paper is available as a technical report which establishes the correctness of the analysis and contains the proofs of the main results.

2 Syntax

We shall study a functional language with side-effects in the style of Standard ML [15]. It has variables $x, f \in \mathsf{Var}$, expressions e and constants c given by:

$$e ::= c \mid x \mid \mathtt{fn}_{\pi}\ x\ \mathtt{=>}\ e \mid \mathtt{fun}_{\pi}\ f\ x\ \mathtt{=>}\ e \mid (e_1\ e_2)^l \mid e_1\ \mathtt{;}\ e_2 \mid \mathtt{ref}_{\varpi}\ e$$
$$\mid\ \ \mathtt{!}e \mid e_1\ \mathtt{:=}\ e_2 \mid \mathtt{let}\ x\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2 \mid \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2$$

$$c ::= \mathtt{true} \mid \mathtt{false} \mid \mathtt{()} \mid \cdots$$

$$
\begin{array}{rl}
m \in \mathsf{Mem} & = \{\diamond\} \ \cup \ (\mathsf{Lab} \times \mathsf{Mem}) \times \widehat{\mathsf{Val}} \times \widehat{\mathsf{Store}} \times (\mathsf{Pnt_F} \times \mathsf{Mem}) \\
d \in \mathsf{Data} & = \cdots \qquad \text{(unspecified)} \\
(\pi, m_d) \in \mathsf{Closure} & = \mathsf{Pnt_F} \times \mathsf{Mem} \\
(\varpi, m_d) \in \mathsf{Cell} & = \mathsf{Pnt_R} \times \mathsf{Mem} \\
v \in \mathsf{Val_A} & = \mathsf{Data} \cup \mathsf{Closure} \cup \mathsf{Cell} \\
W \in \widehat{\mathsf{Val}} & = \mathcal{P}(\mathsf{Mem} \times \mathsf{Val_A}) \\
R \in \widehat{\mathsf{Env}} & = \mathsf{Var} \to \widehat{\mathsf{Val}} \\
S \in \widehat{\mathsf{Store}} & = \mathsf{Cell} \to \widehat{\mathsf{Val}}
\end{array}
$$

Table 1. Abstract domains.

Here $\texttt{fn}_\pi\ x\ \texttt{=>}\ e$ is a function that takes one argument and $\texttt{fun}_\pi\ f\ x\ \texttt{=>}\ e$ is a recursive function (named f) that also takes one argument. We have labelled all syntactic occurrences of function applications with a label $l \in \mathsf{Lab}$, all defining occurrences of functions with a label $\pi \in \mathsf{Pnt_F}$ and all defining occurrences of references with a label $\varpi \in \mathsf{Pnt_R}$.

In Appendix A the semantics is specified as a big-step operational semantics with environments ρ and stores σ. The language has static scope rules and we give it a traditional call-by-value semantics using judgements of the form $\rho \vdash \langle e, \sigma_1 \rangle \to \langle \omega, \sigma_2 \rangle$.

3 Abstract Domains

Mementoes. The analysis will gain its precision from the so-called *mementoes* (or contours or tokens). A memento $m \in \mathsf{Mem}$ represents an approximation of the *context* of a program point: it will either be $\diamond$ representing the initial context where no function calls have taken place or it will have the form

$$((l, m_h), W, S, (\pi, m_d))$$

representing the context in which a function is called. The idea is that

- (l, m_h) describes the application point; l is the label of the function application and m_h is the memento at the application point,
- W is an approximation of the actual parameter at the application point,
- S is an approximation of the store at the application point, and
- (π, m_d) describes the function that is called; π is the label of the function definition and m_d is the memento at the definition point of the function.

Note that this is well-defined (in the manner of context-free grammars): composite mementoes are constructed from simpler mementoes and in the end from the initial memento $\diamond$. This definition of mementoes is akin to the contexts considered in [5]; in Section 5 we shall show how the set can be simplified into something more tractable.

$$\begin{aligned}
\mathcal{R}^d_F, \mathcal{R}^c_F \in \widehat{\mathsf{RCache_F}} &= \mathsf{Pnt_F} \to \widehat{\mathsf{Env}} \\
\mathcal{M}_F \in \widehat{\mathsf{MCache_F}} &= \mathsf{Pnt_F} \to \mathcal{P}(\mathsf{Mem}) \\
\mathcal{W}_F \in \widehat{\mathsf{WCache_F}} &= (\bullet\mathsf{Pnt_F} \cup \mathsf{Pnt_F}\bullet) \to \widehat{\mathsf{Val}} \\
\mathcal{S}_F \in \widehat{\mathsf{SCache_F}} &= (\bullet\mathsf{Pnt_F} \cup \mathsf{Pnt_F}\bullet) \to \widehat{\mathsf{Store}}
\end{aligned}$$

Table 2. Caches.

Example 1. Consider the program "program" defined by:

$$((\mathtt{fn}_x\ \mathtt{x}\ \mathtt{=>}\ ((\mathtt{x}\ \mathtt{x})^1\ (\mathtt{fn}_y\ \mathtt{y}\ \mathtt{=>}\ \mathtt{x}))^2)\ (\mathtt{fn}_z\ \mathtt{z}\ \mathtt{=>}\ \mathtt{z}))^3$$

The applications are performed in the order 3, 1 and 2. The mementoes of interest are going to be: $m_3 = ((3,\diamond), W_3, [\,], (x,\diamond)), m_1 = ((1,m_3), W_1, [\,], (z,\diamond)), m_2 = ((2,m_1), W_2, [\,], (z,\diamond))$ where W_1, W_2 and W_3 will be specified in Example 2 and $[\,]$ indicates that the store is empty. □

Abstract values. We operate on three kinds of abstract values: data, function closures and reference cells. Function closures and reference cells are represented as pairs consisting of the label (π and ϖ, respectively) of the definition point and the memento m_d at the definition point; this will allow us to distinguish between the various instances of the closures and reference cells. The abstract values will always come together with the memento (i.e. the context) in which they live so the analysis will operate over sets of pairs of mementoes and abstract values. The set $\widehat{\mathsf{Val}}$ obtained in this way is equipped with the subset ordering (denoted $\subseteq$). The sets $\widehat{\mathsf{Env}}$ and $\widehat{\mathsf{Store}}$ of abstract environments and abstract stores, respectively, are now obtained in an obvious way and ordered by the pointwise extension of the subset ordering (denoted $\sqsubseteq$).

Example 2. Continuing Example 1 we have

$$W_3 = \{(\diamond, (z,\diamond))\}\ W_1 = \{(m_3, (z,\diamond))\}\ W_2 = \{(m_1, (y, m_3))\}$$

since the function z is defined at the top-level ($\diamond$) and y is defined inside the application 3. □

Caches. The analysis will operate on five caches associating information with functions; their functionality is shown in Table 2. The caches $\mathcal{R}^d_F$, $\mathcal{R}^c_F$ and $\mathcal{M}_F$ associate information with the labels π of function *definitions*:

- The *environment caches* $\mathcal{R}^d_F$ and $\mathcal{R}^c_F$: for each program point π, $\mathcal{R}^d_F(\pi)$ records the abstract environment at the definition point and $\mathcal{R}^c_F(\pi)$ records the same information but modified to each of the contexts in which the function body might be executed. — As an example, the same value v of a variable x used in a function labelled π may turn up in $\mathcal{R}^d_F(\pi)(x)$ as (m_d, v) and in $\mathcal{R}^c_F(\pi)(x)$ as (m_c, v) where $m_d = \diamond$ in case of a top-level function and $m_c = ((l,\diamond), W, S, (\pi,\diamond))$ in case of a top-level application l.

– The *memento cache* $\mathcal{M}_F$: for each program point π, $\mathcal{M}_F(\pi)$ records the set of contexts in which the function body might be executed; so $\mathcal{M}_F(\pi) = \emptyset$ means that the function is never executed.

The caches $\mathcal{W}_F$ and $\mathcal{S}_F$ associate information with function *calls*. For a function with label $\pi \in \mathsf{Pnt_F}$ we shall use $\bullet\pi$ ($\in \bullet\mathsf{Pnt_F}$) to denote the point just before entering the body of the function, and we shall use $\pi\bullet$ ($\in \mathsf{Pnt_F}\bullet$) to denote the point just after leaving the body of the function. The idea now is as follows:

– The *value cache* $\mathcal{W}_F$: for each entry point $\bullet\pi$, $\mathcal{W}_F(\bullet\pi)$ records the abstract value describing the possible actual parameters, and for each exit point $\pi\bullet$, $\mathcal{W}_F(\pi\bullet)$ records the abstract value describing the possible results of the call.
– The *store cache* $\mathcal{S}_F$: for each entry point $\bullet\pi$, $\mathcal{S}_F(\bullet\pi)$ records the abstract store describing the possible stores at function entry, and for each exit point $\pi\bullet$, $\mathcal{S}_F(\pi\bullet)$ records the abstract store describing the possible stores at function exit.

Example 3. For the example program we may take the following caches:

π	x	y	z
$\mathcal{W}_F(\bullet\pi)$	$\{(m_3,(z,\diamond))\}$	$\emptyset$	$\{(m_1,(z,\diamond)),(m_2,(y,m_3))\}$
$\mathcal{W}_F(\pi\bullet)$	$\{(m_3,(y,m_3))\}$	$\emptyset$	$\{(m_1,(z,\diamond)),(m_2,(y,m_3))\}$
$\mathcal{S}_F(\bullet\pi)$	$[\,]$	$[\,]$	$[\,]$
$\mathcal{S}_F(\pi\bullet)$	$[\,]$	$[\,]$	$[\,]$
$\mathcal{R}^d_F(\pi)$	$[\,]$	$[\mathtt{x} \mapsto \{(m_3,(z,\diamond))\}]$	$[\,]$
$\mathcal{R}^c_F(\pi)$	$[\,]$	$[\,]$	$[\,]$
$\mathcal{M}_F(\pi)$	$\{m_3\}$	$\emptyset$	$\{m_1,m_2\}$

4 Syntax-directed Analysis

The specification developed in this section is a recipe for *checking* that a proposed solution is indeed acceptable. This is useful when changing libraries of support code or when installing software in new environments: one merely needs to check that the new libraries or environments satisfy the solution used to optimise the program. It can also be used as the basis for generating a set of constraints [17] whose least solution can be obtained using standard techniques (e.g. [2]).

Given a program e and the five caches $(\mathcal{R}^d_F, \mathcal{R}^c_F, \mathcal{M}_F, \mathcal{W}_F, \mathcal{S}_F)$ the purpose of the analysis is to check whether or not the caches are acceptable solutions to the Data and Control Flow Analysis. The first step is to find (or guess) the following auxiliary information:

– an abstract environment $R \in \widehat{\mathsf{Env}}$ describing the free variables in e (and typically it is $\perp$ if there are no free variables in the program),

- a set of mementoes $M \in \mathcal{P}(\mathsf{Mem})$ describing the possible contexts in which e can be evaluated (and typically it is $\{\diamond\}$),
- an initial abstract store $S_1 \in \widehat{\mathsf{Store}}$ describing the mutable store before evaluation of e begins (and typically it is $\top$ if the store is not initialised before use),
- a final abstract store $S_2 \in \widehat{\mathsf{Store}}$ describing the mutable store after evaluation of e completes (and possibly it is $\top$), and
- an abstract value $W \in \widehat{\mathsf{Val}}$ describing the value that e can evaluate to (and it also possibly is $\top$).

The second step is to check whether or not the formula

$$R, M \ \rhd \ e : S_1 \to S_2 \ \& \ W$$

is satisfied with respect to the caches supplied. This means that when e is executed in an environment described by R, in a context described by M, and upon a state described by S_1 the following happens: if e terminates successfully then the resulting state is described by S_2 and the resulting value by W.

We shall first specify the analysis for the functional fragment of the language (Table 3) and then for the other constructs (Table 4). As in [16] any free variable on the right-hand side of the clauses should be regarded as existentially quantified; in principle this means that their values need to be guessed, but in practice the best (or least) guess mostly follows from the subformulae.

Example 4. Given the caches of Example 3, we shall check the formula:

$$[\,], \{\diamond\} \rhd \mathsf{program} : [\,] \to [\,] \ \& \ \{(\diamond, (y, m_3))\}$$

So the initial environment is empty, the initial context is $\diamond$, the program does not manipulate the store, and the final value is described by $\{(\diamond, (y, m_3))\}$. □

The functional fragment. For all five constructs in the functional fragment of the language the handling of the store is straightforward since it is threaded in the same way as in the semantics.

For constants and variables it is fairly straightforward to determine the abstract value for the construct; in the case of variables we obtain it from the environment and in the other case we construct it from the set M of mementoes of interest.

For function definitions no changes need take place in the store so the abstract store is simply threaded as in the previous cases. The abstract value representing the function definition contains a nested pair (a triple) for each memento m in the set M of mementoes according to which the function definition can be reached: in a nested pair $(m_1, (\pi, m_2))$ the memento m_1 represents the current context and the pair (π, m_2) represents the value produced (and demanding that $m_1 = m_2$ corresponds to performing a precise relational analysis rather than a

$$R, M \ \rhd \ c : S_1 \to S_2 \ \& \ W$$
$$\quad \textsf{iff} \ S_1 \sqsubseteq S_2 \wedge \{(m, d_c) \mid m \in M\} \subseteq W$$

$$R, M \ \rhd \ x : S_1 \to S_2 \ \& \ W$$
$$\quad \textsf{iff} \ S_1 \sqsubseteq S_2 \wedge R(x) \subseteq W$$

$$R, M \ \rhd \ \texttt{fn}_\pi \ x \ \texttt{=>} \ e : S_1 \to S_2 \ \& \ W$$
$$\quad \textsf{iff} \ S_1 \sqsubseteq S_2 \wedge \{(m, (\pi, m)) \mid m \in M\} \subseteq W \wedge R \sqsubseteq \mathcal{R}^d_F(\pi) \wedge$$
$$\quad\quad \mathcal{R}^c_F(\pi)[x \mapsto \mathcal{W}_F(\bullet\pi)], \mathcal{M}_F(\pi) \ \rhd \ e : \mathcal{S}_F(\bullet\pi) \to \mathcal{S}_F(\pi\bullet) \ \& \ \mathcal{W}_F(\pi\bullet)$$

$$R, M \ \rhd \ \texttt{fun}_\pi \ f \ x \ \texttt{=>} \ e : S_1 \to S_2 \ \& \ W$$
$$\quad \textsf{iff} \ S_1 \sqsubseteq S_2 \wedge \{(m, (\pi, m)) \mid m \in M\} \subseteq W \wedge$$
$$\quad\quad R[f \mapsto \{(m, (\pi, m)) \mid m \in M\}] \sqsubseteq \mathcal{R}^d_F(\pi) \wedge$$
$$\quad\quad \mathcal{R}^c_F(\pi)[x \mapsto \mathcal{W}_F(\bullet\pi)], \mathcal{M}_F(\pi) \ \rhd \ e : \mathcal{S}_F(\bullet\pi) \to \mathcal{S}_F(\pi\bullet) \ \& \ \mathcal{W}_F(\pi\bullet)$$

$$R, M \ \rhd \ (e_1 \ e_2)^l : S_1 \to S_4 \ \& \ W$$
$$\quad \textsf{iff} \ R, M \ \rhd \ e_1 : S_1 \to S_2 \ \& \ W_1 \ \wedge R, M \ \rhd \ e_2 : S_2 \to S_3 \ \& \ W_2 \ \wedge$$
$$\quad\quad \forall \pi \in \{\pi \mid (m, (\pi, m_d)) \in W_1\} :$$
$$\quad\quad\quad \textsf{let} \ X = \overline{\textsf{new}}_\pi((l, M), W_2, S_3, W_1)$$
$$\quad\quad\quad\quad X_{dc} = \{(m_d, m_c) \mid (m_d, m_h, m_c) \in X\}$$
$$\quad\quad\quad\quad X_c \ = \{m_c \mid (m_d, m_h, m_c) \in X\}$$
$$\quad\quad\quad\quad X_{hc} = \{(m_h, m_c) \mid (m_d, m_h, m_c) \in X\}$$
$$\quad\quad\quad\quad X_{ch} = \{(m_c, m_h) \mid (m_d, m_h, m_c) \in X\}$$
$$\quad\quad\quad \textsf{in} \ \mathcal{R}^d_F(\pi)\lceil X_{dc} \rceil \sqsubseteq \mathcal{R}^c_F(\pi) \wedge X_c \subseteq \mathcal{M}_F(\pi) \wedge$$
$$\quad\quad\quad\quad W_2\lceil X_{hc} \rceil \subseteq \mathcal{W}_F(\bullet\pi) \wedge S_3\lceil X_{hc} \rceil \sqsubseteq \mathcal{S}_F(\bullet\pi) \wedge$$
$$\quad\quad\quad\quad \mathcal{W}_F(\pi\bullet)\lceil X_{ch} \rceil \subseteq W \wedge \mathcal{S}_F(\pi\bullet)\lceil X_{ch} \rceil \sqsubseteq S_4$$

$$\overline{\textsf{new}}_\pi((l, M), W, S, W') =$$
$$\quad \{(m_d, m_h, m_c) \mid (m_h, (\pi, m_d)) \in W', \ m_h \in M, \ m_c = \textsf{new}((l, m_h), W, S, (\pi, m_d))\}$$

Table 3. Analysis of the functional fragment.

less precise independent attribute analysis). Finally, the body of the function is analysed in the relevant abstract environment, memento set, initial abstract state, final abstract state and final abstract value; this information is obtained from the caches that are in turn updated at the corresponding call points. More precisely, the idea is to record the abstract environment at the definition point in the cache $\mathcal{R}^d_F$ and then to analyse the body of the function in the context of the call which is specified by the caches $\mathcal{R}^c_F$, $\mathcal{M}_F$, $\mathcal{W}_F$ and $\mathcal{S}_F$ as explained in Section 3. The clause for recursive functions is similar.

Example 5. To check the formula of Example 4 we need among other things to check:

$$[\,], \{\diamond\} \rhd \texttt{fn}_z \ \texttt{z => z} : [\,] \to [\,] \ \& \ \{(\diamond, (z, \diamond))\}$$

This follows from the clause for function definition because $[\,] \sqsubseteq [\,]$ and the clause for variables gives:

$$[\texttt{z} \mapsto \{(m_1, (z, \diamond)), (m_2, (y, m_3))\}], \{m_1, m_2\} \rhd \texttt{z} : [\,] \to [\,] \ \& \ \{(m_1, (z, \diamond)), (m_2, (y, m_3))\}$$

Note that although the function z is *called twice*, it is only *analysed once.* □

In the clause for the function application $(e_1\ e_2)^l$ we first analyse the operator and the operand while threading the store. Then we use W_1 to determine which functions can be called and for each such function π we proceed in the following way.

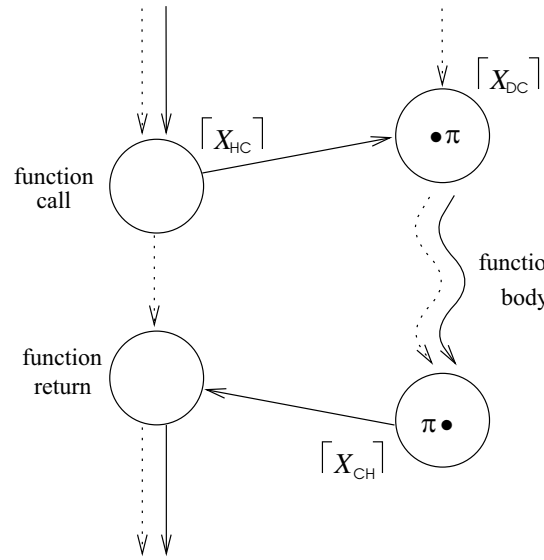


Fig. 2. Analysis of function call.

First we determine the mementoes to be used for analysing the body of the function π. More precisely we calculate a set X of triples (m_d, m_h, m_c) consisting of a definition memento m_d describing the point where the function π was defined, a current memento m_h describing the call point, and a memento m_c describing the entry point to the procedure body. (For the call $(\mathtt{x}\ \mathtt{x})^1$ in Example 1 we would have $X = \{(\diamond, m_3, m_1)\}$ and $\pi = z$.) For this we use the operation $\overline{\mathsf{new}}_\pi$ whose definition (see Table 3) uses the function

$$\mathsf{new} : (\mathsf{Lab} \times \mathsf{Mem}) \times \widehat{\mathsf{Val}} \times \widehat{\mathsf{Store}} \times (\mathsf{Pnt}_\mathsf{F} \times \mathsf{Mem}) \to \mathsf{Mem}$$

for converting its argument to a memento. With Mem defined as in Table 1 this will be the identity function but for simpler choices of Mem it will discard some of the information supplied by its argument.

The sets X_{dc}, X_c, X_{hc}, and X_{ch} are "projections" of X. The body of the function π will be analysed in the set of mementoes obtained as X_c and therefore X_c is recorded in the cache $\mathcal{M}_F$ for use in the clause defining the function. Because the function body is analysed in this set of mementoes we need to modify the mementoes components of all the relevant abstract values. For this we use the operation

$$W\lceil Y \rceil = \{(m_2, v) \mid (m_1, v) \in W, (m_1, m_2) \in Y\}$$

defined on $W \subseteq \widehat{\mathsf{Val}}$ and $Y \subseteq \mathsf{Mem} \times \mathsf{Mem}$. This operation is lifted to abstract environments and abstract stores in a pointwise manner.

Coming back to the clause for application in Table 3, the abstract environment $\mathcal{R}_F^d(\pi)$ is relative to the mementoes of the definition point for the function and thus has to be modified so as to be relative to the mementoes of the called function body and the set X_{dc} facilitates performing this transformation. (For the call $(\mathtt{x}\ \mathtt{x})^1$ in Example 1 we would have that $X_{dc} = \{(\diamond, m_1)\}$.) In this way we ensure that we have static scoping of the free variables of the function. The actual parameter W_2 is relative to the mementoes of the application point and has to be modified so as to be relative to the mementoes of the called function body and the set X_{hc} facilitates performing this transformation; a similar modification is needed for the abstract store at the entry point. We also need to link the results of the analysis of the function body back to the application point and here the relevant transformation is facilitated by the set X_{ch}.

The clause for application is illustrated in Figure 2. On the left-hand side we have the application point with explicit nodes for the call and the return. The dotted lines represent the abstract environment and the relevant set of mementoes whereas the solid lines represent the values (actual parameter and result) and the store. The transfer function $\lceil X_{dc} \rceil$ is used to modify the static environment of the definition point, the transfer function $\lceil X_{hc} \rceil$ is used to go from the application point to the function body and the transfer function $\lceil X_{ch} \rceil$ is used to go back from the function body to the application point. Note that the figure clearly indicates the different paths taken by environment-like information and store-like information – something that is not always clear from similar figures appearing in the literature (see Section 5).

Example 6. Checking the formula of Example 4 also involves checking:

$$[\mathtt{x} \mapsto \{(m_3, (z, \diamond))\}], \{m_3\} \rhd (\mathtt{x}\ \mathtt{x})^1 : [\,] \to [\,] \ \&\ \{(m_3, (z, \diamond))\}$$

For this, the clause for application demands that we check

$$[\mathtt{x} \mapsto \{(m_3, (z, \diamond))\}], \{m_3\} \rhd \mathtt{x} : [\,] \to [\,] \ \&\ \{(m_3, (z, \diamond))\}$$

which follows directly from the clause for variables.

Only the function z can be called so we have to check the many conditions only for this function. We shall concentrate on checking that $\{(m_3, (z, \diamond))\}\lceil X_{hc} \rceil \subseteq \mathcal{W}_F(\bullet z)$ and $\mathcal{W}_F(z\bullet)\lceil X_{ch} \rceil \subseteq \{(m_3, (z, \diamond))\}$. Since $X = \{(\diamond, m_3, m_1)\}$ we have $X_{hc} = \{(m_3, m_1)\}$ and the effect of the transformation will be to remove all pairs that do not have m_3 as the first component and to replace the first components of the remaining pairs with m_1; using Example 3 it is immediate to verify that the condition actually holds. Similarly, $X_{ch} = \{(m_1, m_3)\}$ so in this case the transformation will remove pairs that do not have m_1 as the first component (i.e. pairs that do not correspond to the current call point) and replace the first components of the remaining pairs with m_3; again it is immediate to verify that the condition holds. □

Other constructs. The clauses for the other constructs of the language are shown in Table 4. The clauses reflect that the abstract environment and the set

$$R, M \ \rhd \ e_1 \ ; \ e_2 : S_1 \to S_3 \ \& \ W_2$$
$$\quad \textsf{iff } R, M \ \rhd \ e_1 : S_1 \to S_2 \ \& \ W_1 \wedge R, M \ \rhd \ e_2 : S_2 \to S_3 \ \& \ W_2$$

$$R, M \ \rhd \ \texttt{ref}_{\varpi} \ e : S_1 \to S_3 \ \& \ W'$$
$$\quad \textsf{iff } R, M \ \rhd \ e : S_1 \to S_2 \ \& \ W \wedge \{(m, (\varpi, m)) \mid m \in M\} \subseteq W' \wedge S_2 \sqsubseteq S_3 \wedge$$
$$\qquad \forall m \in M : W \subseteq S_3(\varpi, m)$$

$$R, M \ \rhd \ \texttt{!}e : S_1 \to S_2 \ \& \ W'$$
$$\quad \textsf{iff } R, M \ \rhd \ e : S_1 \to S_2 \ \& \ W \wedge \forall (m, (\varpi, m_d)) \in W : S_2(\varpi, m_d) \subseteq W'$$

$$R, M \ \rhd \ e_1 \ \texttt{:=} \ e_2 : S_1 \to S_4 \ \& \ W$$
$$\quad \textsf{iff } R, M \ \rhd \ e_1 : S_1 \to S_2 \ \& \ W_1 \wedge R, M \ \rhd \ e_2 : S_2 \to S_3 \ \& \ W_2 \wedge$$
$$\qquad \{(m, d_{()}) \mid m \in M\} \subseteq W \wedge S_3 \sqsubseteq S_4 \wedge \forall (m, (\varpi, m_d)) \in W_1 : W_2 \subseteq S_4(\varpi, m_d)$$

$$R, M \ \rhd \ \texttt{let} \ x \ \texttt{=} \ e_1 \ \texttt{in} \ e_2 : S_1 \to S_3 \ \& \ W_2$$
$$\quad \textsf{iff } R, M \ \rhd \ e_1 : S_1 \to S_2 \ \& \ W_1 \wedge R[x \mapsto W_1], M \ \rhd \ e_2 : S_2 \to S_3 \ \& \ W_2$$

$$R, M \ \rhd \ \texttt{if} \ e \ \texttt{then} \ e_1 \ \texttt{else} \ e_2 : S_1 \to S_5 \ \& \ W'$$
$$\quad \textsf{iff } R, M \ \rhd \ e : S_1 \to S_2 \ \& \ W \wedge$$
$$\qquad \textsf{let } R_1 = \varphi^{[e,W]}_{\texttt{true}}(R); \quad R_2 = \varphi^{[e,W]}_{\texttt{false}}(R); \quad S_3 = \phi^{[e,W]}_{\texttt{true}}(S_2); \quad S_4 = \phi^{[e,W]}_{\texttt{false}}(S_2)$$
$$\qquad \textsf{in } \ R_1, M \ \rhd \ e_1 : S_3 \to S_5 \ \& \ W' \ \wedge R_2, M \ \rhd \ e_2 : S_4 \to S_5 \ \& \ W'$$

Table 4. Analysis of the other constructs.

of mementoes are passed to the subexpressions in a syntax-directed way and that the store is threaded through the constructs. The analysis is fairly simple-minded in that it does not try to predict when a reference (ϖ, m_d) in the analysis only represents one location in the semantics and hence the analysis does not contain any kill-components (but see Appendix B).

For the `let`-construct we perform the expected threading of the abstract environment and the abstract store. For the conditional we first analyse the condition. Based on the outcome we then modify the environment and the store to reflect the (abstract) value of the test. For the environment we use the transfer functions $\varphi^{[e,W]}_{\texttt{true}}(R)$ and $\varphi^{[e,W]}_{\texttt{false}}(R)$ whereas for the store we use the transfer functions $\phi^{[e,W]}_{\texttt{true}}(S_2)$ and $\phi^{[e,W]}_{\texttt{false}}(S_2)$. The result of both branches are possible for the whole construct.

As an example of the use of these transfer functions consider the expression `if` x `then` e_1 `else` e_2 where it will be natural to set

$$\varphi^{[x,W]}_{\texttt{true}}(R) = R[x \mapsto W \cap \{(m, d_{\texttt{true}}) \mid m \in \mathsf{Mem}\}]$$

and similarly for $\varphi^{[x,W]}_{\texttt{false}}(R)$. Thus it will be possible to analyse each of the branches with precise information about x.

Little can be said in general about how to define such functions; to obtain a more concise statement of the theorems below we shall *assume* that the transfer functions $\phi^{\cdots}_{\cdots}$ and $\varphi^{\cdots}_{\cdots}$ of Table 4 are in fact the identities.

$$\begin{array}{rl} m_k \in \mathsf{Mem}_k & = \mathsf{Lab}^{\leq k} \\ d \in \mathsf{Data} & = \cdots \qquad \text{(unspecified)} \\ (\pi, m_{kd}) \in \mathsf{Closure}_k & = \mathsf{Pnt_F} \times \mathsf{Mem}_k \\ (\varpi, m_{kd}) \in \mathsf{Cell}_k & = \mathsf{Pnt_R} \times \mathsf{Mem}_k \\ v_k \in \mathsf{Val}_{\mathsf{A}k} & = \mathsf{Data} \cup \mathsf{Closure}_k \cup \mathsf{Cell}_k \\ W_k \in \widehat{\mathsf{Val}}_k & = \mathcal{P}(\mathsf{Mem}_k \times \mathsf{Val}_{\mathsf{A}k}) \\ R_k \in \widehat{\mathsf{Env}}_k & = \mathsf{Var} \to \widehat{\mathsf{Val}}_k \\ S_k \in \widehat{\mathsf{Store}}_k & = \mathsf{Cell}_k \to \widehat{\mathsf{Val}}_k \end{array}$$

Table 5. Abstract domains for k-CFA.

5 Classical Approximations

***k*-CFA.** The idea behind k-CFA [11,24] is to restrict the mementoes to keep track of the last k call sites only. This leads to the abstract domains of Table 5 that are intended to replace Table 1. Naturally, the analysis of Tables 2, 3, and 4 must be modified to use the new abstract domains; also the function $\overline{\mathsf{new}}_\pi$ must be modified to make use of the function

$$\mathsf{new}_k : (\mathsf{Lab} \times \mathsf{Mem}_k) \times \widehat{\mathsf{Val}}_k \times \widehat{\mathsf{Store}}_k \times (\mathsf{Pnt_F} \times \mathsf{Mem}_k) \to \mathsf{Mem}_k$$

defined by $\mathsf{new}_k((l, m_{kh}), W_k, S_k, (\pi, m_{kd})) = \mathsf{take}_k(l \hat{} m_{kh})$ where "$\hat{}$" denotes prefixing and take_k returns the first k elements of its argument. This completes the definition of the analysis.

Theoretical properties. One of the strong points of our approach is that we can use the framework of Abstract Interpretation to describe *how* the more tractable choices of mementoes arise from the general definition.

To express the relationship between the two analyses define a *surjective* mapping $\mu_k : \mathsf{Mem} \to \mathsf{Mem}_k$ showing how the precise mementoes of Table 1 are truncated into the approximative mementoes of Table 5. It is defined by $\mu_0(m) = \varepsilon, \mu_{k+1}(\diamond) = \varepsilon, \mu_{k+1}((l, m), W, S, (\pi, m_d)) = l \hat{} \mu_k(m)$ where ε denotes the empty sequence. It gives rise to the functions $\alpha_k^M : \mathcal{P}(\mathsf{Mem}) \to \mathcal{P}(\mathsf{Mem}_k)$ and $\gamma_k^M : \mathcal{P}(\mathsf{Mem}_k) \to \mathcal{P}(\mathsf{Mem})$ defined by $\alpha_k^M(M) = \{\mu_k(m) \mid m \in M\}$ and $\gamma_k^M(M_k) = \{m \mid \mu_k(m) \in M_k\}$. Since α_k^M is surjective and defined in a pointwise manner there exists precisely one function such that

$$\mathcal{P}(\mathsf{Mem}) \underset{\alpha_k^M}{\overset{\gamma_k^M}{\leftrightarrows}} \mathcal{P}(\mathsf{Mem}_k)$$

is a *Galois insertion* as studied in Abstract Interpretation [4]: this means that α_k^M and γ_k^M are both monotone and that $\gamma_k^M(\alpha_k^M(M)) \supseteq M$ and $\alpha_k^M(\gamma_k^M(M_k)) = M_k$ for all $M \subseteq \mathsf{Mem}$ and $M_k \subseteq \mathsf{Mem}_k$. One may check that γ_k^M is as displayed above.

To obtain a Galois insertion

$$\widehat{\mathsf{Val}} \underset{\alpha_k^V}{\overset{\gamma_k^V}{\leftrightarrows}} \widehat{\mathsf{Val}}_k$$

we first define a surjective mapping $\eta_k : \mathsf{Mem} \times \mathsf{Val_A} \to \mathsf{Mem}_k \times \mathsf{Val_A}k$ by taking $\eta_k(m_h, d) = (\mu_k(m_h), d)$, $\eta_k(m_h, (\pi, m_d)) = (\mu_k(m_h), (\pi, \mu_k(m_d)))$, and $\eta_k(m_h, (\varpi, m_d)) = (\mu_k(m_h), (\varpi, \mu_k(m_d)))$. Next define α_k^V and γ_k^V by $\alpha_k^V(W) = \{\eta_k(m, v) \mid (m, v) \in W\}$ and $\gamma_k^V(W_k) = \{(m, v) \mid \eta_k(m, v) \in W_k\}$. It is then straightforward to obtain a Galois insertion

$$\widehat{\mathsf{Env}} \underset{\alpha_k^E}{\overset{\gamma_k^E}{\leftrightarrows}} \widehat{\mathsf{Env}}_k$$

by setting $\alpha_k^E(R)(x) = \alpha_k^V(R(x))$ and $\gamma_k^E(R_k)(x) = \gamma_k^V(R_k(x))$. To obtain a Galois insertion

$$\widehat{\mathsf{Store}} \underset{\alpha_k^S}{\overset{\gamma_k^S}{\leftrightarrows}} \widehat{\mathsf{Store}}_k$$

define $\alpha_k^S(S)(\varpi, m_{kd}) = \alpha_k^V(\bigcup\{S(\varpi, m_d) \mid \mu_k(m_d) = m_{kd}\})$ and $\gamma_k^S(S_k)(\varpi, m_d) = \gamma_k^V(S_k(\varpi, \mu_k(m_d)))$.

We now have the machinery needed to state the relationship between the present k-CFA analysis (denoted $\rhd_k$) and the general analysis of Section 4 (denoted $\rhd$):

Theorem 1. *If* $(\mathcal{R}^d_{kF}, \mathcal{R}^c_{kF}, \mathcal{M}_{kF}, \mathcal{W}_{kF}, \mathcal{S}_{kF})$ *satisfies*

$$R_k, M_k \ \rhd_k \ e : S_{k1} \to S_{k2} \ \&\ W_k$$

then $(\gamma_k^E \circ \mathcal{R}^d_{kF}, \gamma_k^E \circ \mathcal{R}^c_{kF}, \gamma_k^M \circ \mathcal{M}_{kF}, \gamma_k^V \circ \mathcal{W}_{kF}, \gamma_k^S \circ \mathcal{S}_{kF})$ *satisfies*

$$\gamma_k^E(R_k), \gamma_k^M(M_k) \ \rhd \ e : \gamma_k^S(S_{k1}) \to \gamma_k^S(S_{k2}) \ \&\ \gamma_k^V(W_k).$$

In the full version we establish the semantic correctness of the analysis of Section 4; it then follows that semantic correctness holds for k-CFA as well.

Call strings of length *k*. The clause for application involves a number of transfers using the set X relating definition mementoes, current mementoes and mementoes of the called function body. In the case of a k-CFA like approach it may be useful to simplify these transfers.

The transfer using X_{hc} can be implemented in a simple way by taking

$$X_{hc} = \{(m_h, \mathsf{take}_k(l\hat{}m_h)) \mid m_h \in M\}$$

where l is the label of the application point. This set may be slightly too large because it is no longer allowed to depend on the actual function called (the π) and because there may be $m_h \in M_k$ for which no $(m_h, (\pi, m_d))$ is ever an

element of W_1. However, this is just a minor imprecision aimed at facilitating a more efficient implementation. In a similar way, one may take

$$X_c = \{\mathsf{take}_k(l\hat{}m_h) \mid m_h \in M\}$$

where again this set may be slightly too large.

The transfers using X_{ch} can also be somewhat simplified by taking

$$\begin{aligned} X_{ch} &= \{(\mathsf{take}_k(l\hat{}m_h), m_h) \mid m_h \in M\} \\ &= \{(m_c, \mathsf{drop}_1(m_c)) \mid \mathsf{drop}_1(m_c) \in M\} \\ &\quad \cup \{(m_c, \mathsf{drop}_1(m_c)\hat{}l') \mid \mathsf{drop}_1(m_c)\hat{}l' \in M\} \end{aligned}$$

where drop_1 drops the first element of its argument (yielding ε if the argument does not have at least two elements). Again this set may be slightly too large.

The transfer using X_{dc} can be rewritten as

$$X_{dc} = \{(m_d, \mathsf{take}_k(l\hat{}m_h)) \mid m_h \in M, (m_h, (\pi, m_d)) \in W_1\}$$

where l is the application point and π is the function called.

For functions being defined at top-level there is not likely to be too much information that need to be transformed using X_{dc}; however, simplifying X_{dc} to be independent of π is likely to be grossly imprecise.

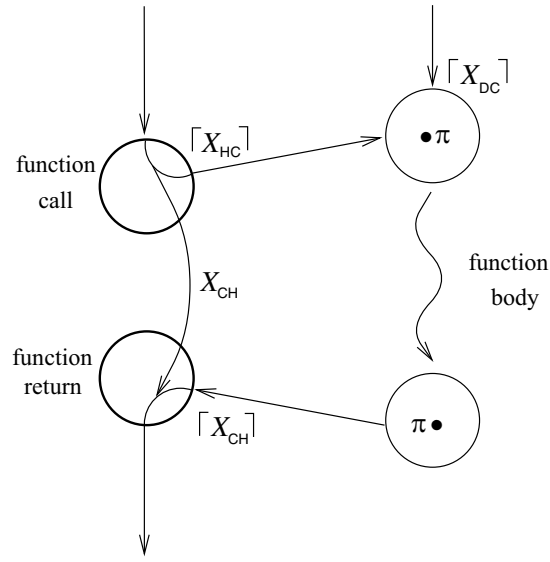


Fig. 3. Degenerate analysis of function call.

Performing these modifications to the clause for application there is no longer any need for an explicit call of $\overline{\mathsf{new}}_\pi$. The resulting analysis is similar in spirit to the call string based analysis of [27]; the scenario of [23] is simpler because the language considered there does not allow local data. Since we have changed the definition of the sets X_{dc}, X_c, X_{hc} and X_{ch} to something that is no less than before, it follows that an analogue of Theorem 1 still applies and therefore the semantic correctness result still carries over.

$$
\begin{array}{rl}
m_p \in \mathsf{Mem}_p & = \{\varepsilon\} \ \cup \ \mathcal{P}(\mathsf{Data} \cup \mathsf{Pnt_F} \cup \mathsf{Pnt_R}) \\
d \in \mathsf{Data} & = \cdots \qquad \text{(unspecified)} \\
(\pi, m_{pd}) \in \mathsf{Closure}_p & = \mathsf{Pnt_F} \times \mathsf{Mem}_p \\
(\varpi, m_{pd}) \in \mathsf{Cell}_p & = \mathsf{Pnt_R} \times \mathsf{Mem}_p \\
v_p \in \mathsf{Val}_{\mathsf{A}p} & = \mathsf{Data} \cup \mathsf{Closure}_p \cup \mathsf{Cell}_p \\
W_p \in \widehat{\mathsf{Val}}_p & = \mathcal{P}(\mathsf{Mem}_p \times \mathsf{Val}_{\mathsf{A}p}) \\
R_p \in \widehat{\mathsf{Env}}_p & = \mathsf{Var} \to \widehat{\mathsf{Val}}_p \\
S_p \in \widehat{\mathsf{Store}}_p & = \mathsf{Cell}_p \to \widehat{\mathsf{Val}}_p
\end{array}
$$

Table 6. Abstract domains for assumption sets.

It is interesting to note that if the distinction between environment and store is not clearly maintained then Figure 2 degenerates to the form of Figure 3; this is closely related to the scenario in [22] (that is somewhat less general).

Assumption sets. The idea behind this analysis is to restrict the mementoes to keep track of the parameter of the last function call only; such information is often called assumption sets. This leads to the abstract domains of Table 6 that are intended to replace Table 1. Naturally, the analysis of Tables 2, 3, and 4 must be modified to use the new abstract domains; also the function $\overline{\mathsf{new}}_\pi$ must be modified to make use of the function

$$\mathsf{new}_p : (\mathsf{Lab} \times \mathsf{Mem}_p) \times \widehat{\mathsf{Val}}_p \times \widehat{\mathsf{Store}}_p \times (\mathsf{Pnt_F} \times \mathsf{Mem}_p) \to \mathsf{Mem}_p$$

given by $\mathsf{new}_p((l, m_{ph}), W_p, S_p, (\pi, m_{pd})) = \{\mathsf{keep}_p(v_p) \mid (m_p, v_p) \in W_p\}$ where $\mathsf{keep}_p : \mathsf{Val}_{\mathsf{A}p} \to (\mathsf{Data} \cup \mathsf{Pnt_F} \cup \mathsf{Pnt_R})$ is given by $\mathsf{keep}_p(d) = d$, $\mathsf{keep}_p(\pi, m_{pd}) = \pi$, $\mathsf{keep}_p(\varpi, m_{pd}) = \varpi$.

Theoretical properties. We can now mimic the development performed above. The crucial point is the definition of a *surjective* mapping $\mu_p : \mathsf{Mem} \to \mathsf{Mem}_p$ showing how the precise mementoes of Table 1 are mapped into the approximative mementoes of Table 6. It is given by $\mu_p(\diamond) = \varepsilon$, and $\mu_p((l, m), W, S, (\pi, m_d)) = \{\mathsf{keep}'_p(v) \mid (m', v') \in W\}$. where $\mathsf{keep}'_p : \mathsf{Val_A} \to (\mathsf{Data} \cup \mathsf{Pnt_F} \cup \mathsf{Pnt_R})$ is the obvious modification of keep_p to work on $\mathsf{Val_A}$ rather than $\mathsf{Val}_{\mathsf{A}p}$. Based on μ_p we can now define Galois *insertions*

- (α^M_p, γ^M_p) between $\mathcal{P}(\mathsf{Mem})$ and $\mathcal{P}(\mathsf{Mem}_p)$
- (α^V_p, γ^V_p) between $\widehat{\mathsf{Val}}$ and $\widehat{\mathsf{Val}}_p$
- (α^E_p, γ^E_p) between $\widehat{\mathsf{Env}}$ and $\widehat{\mathsf{Env}}_p$
- (α^S_p, γ^S_p) between $\widehat{\mathsf{Store}}$ and $\widehat{\mathsf{Store}}_p$

very much as before and obtain the following analogue of Theorem 1:

Theorem 2. *If* $(\mathcal{R}^d_{pF}, \mathcal{R}^c_{pF}, \mathcal{M}_{pF}, \mathcal{W}_{pF}, \mathcal{S}_{pF})$ *satisfies*

$$R_p, M_p \ \rhd_p \ e : S_{p1} \to S_{p2} \ \& \ W_p$$

then $(\gamma^E_p \circ \mathcal{R}^d_{pF}, \gamma^E_p \circ \mathcal{R}^c_{pF}, \gamma^M_p \circ \mathcal{M}_{pF}, \gamma^V_p \circ \mathcal{W}_{pF}, \gamma^S_p \circ \mathcal{S}_{pF})$ *satisfies*

$$\gamma^E_p(R_p), \gamma^M_p(M_p) \ \rhd \ e : \gamma^S_p(S_{p1}) \to \gamma^S_p(S_{p2}) \ \& \ \gamma^V_p(W_p).$$

As before it is a consequence of the above theorem that semantic correctness holds for the assumption set analysis as well.

6 Conclusion

We have shown how to express interprocedural and context-sensitive Data Flow Analysis in a syntax-directed framework that is reminiscent of Control Flow Analysis; thereby we have not only extended the ability of Data Flow Analysis to deal with higher-order functions but we also have extended the ability of Control Flow Analysis to deal with mutable data structures. At the same time we have used Abstract Interpretation to pass from the general mementoes of Section 3 to the more tractable mementoes of Section 5. In fact *all* our analyses are based on the specification of Tables 3 and 4.

Acknowledgement. This work has been supported in part by the DART project funded by the Danish Science Research Council.

References

1. F. Bourdoncle. Interprocedural abstract interpretation of block structured languages with nested procedures, aliasing and recursivity. In *Proc. PLILP '90*, volume 456 of *Lecture Notes in Computer Science*, pages 307–323. Springer, 1990.
2. F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proc. Formal Methods in Programming and Their Applications*, volume 735 of *Lecture Notes in Computer Science*, pages 128–141. Springer, 1993.
3. P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E. J. Neuhold, editor, *Formal Description of Programming Concepts*. North Holland, 1978.
4. P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proc. POPL '79*, pages 269–282, 1979.
5. A. Deutsch. On Determining Lifetime and Aliasing of Dynamically Allocated Data in Higher Order Functional Specifications. In *Proc. POPL '90*, pages 157–169. ACM Press, 1990.
6. K. L. S. Gasser, F. Nielson, and H. R. Nielson. Systematic realisation of control flow analyses for CML. In *Proc. ICFP '97*, pages 38–51. ACM Press, 1997.
7. M. S. Hecht. *Flow Analysis of Computer Programs*. North Holland, 1977.

8. N. Heintze. Set-based analysis of ML programs. In *Proc. LFP '94*, pages 306–317, 1994.
9. N. Heintze and J. Jaffar. An engine for logic program analysis. In *Proc. LICS '92*, pages 318–328, 1992.
10. S. Jagannathan and S. Weeks. Analyzing Stores and References in a Parallel Symbolic Language. In *Proc. LFP '94*, pages 294–305, 1994.
11. S. Jagannathan and S. Weeks. A unified treatment of flow analysis in higher-order languages. In *Proc. POPL '95*. ACM Press, 1995.
12. N. D. Jones and S. S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Proc. POPL '82*, pages 66–74. ACM Press, 1982.
13. J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *Proc. CC '92*, volume 641 of *Lecture Notes in Computer Science*, pages 125–140. Springer, 1992.
14. W. Landi and B. G. Ryder. Pointer-Induced Aliasing: A Problem Classification. In *Proc. POPL '91*, pages 93–103. ACM Press, 1991.
15. R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML.* MIT Press, 1990.
16. F. Nielson and H. R. Nielson. Infinitary Control Flow Analysis: a Collecting Semantics for Closure Analysis. In *Proc. POPL '97*. ACM Press, 1997.
17. H. R. Nielson and F. Nielson. Flow logics for constraint based analysis. In *Proc. CC '98*, volume 1383 of *Lecture Notes in Computer Science*, pages 109–127. Springer, 1998.
18. J. Palsberg. Global program analysis in constraint form. In *Proc. CAAP '94*, volume 787 of *Lecture Notes in Computer Science*, pages 255–265. Springer, 1994.
19. J. Palsberg. Closure analysis in constraint form. *ACM TOPLAS*, 17 (1):47–62, 1995.
20. H. D. Pande and B. G. Ryder. Data-flow-based virtual function resolution. In *Proc. SAS '96*, volume 1145 of *Lecture Notes in Computer Science*, pages 238–254. Springer, 1996.
21. E. Ruf. Context-insensitive alias analysis reconsidered. In *Proc. PLDI '95*, pages 13–22. ACM Press, 1995.
22. M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. In *Proc. TAPSOFT '95*, volume 915 of *Lecture Notes in Computer Science*, pages 651–665, 1995.
23. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis*. Prentice Hall International, 1981.
24. O. Shivers. Control flow analysis in Scheme. In *Proc. PLDI '88*, volume 7 (1) of *ACM SIGPLAN Notices*, pages 164–174. ACM Press, 1988.
25. O. Shivers. The semantics of Scheme control-flow analysis. In *Proc. PEPM '91*, volume 26 (9) of *ACM SIGPLAN Notices*. ACM Press, 1991.
26. A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5:285–309, 1955.
27. J. Vitek, R. N. Horspool, and J. S. Uhl. Compile-Time Analysis of Object-Oriented Programs. In *Proc. CC '92*, volume 641 of *Lecture Notes in Computer Science*, pages 236–250. Springer, 1992.

$$\rho \vdash \langle c, \sigma\rangle \to \langle c, \sigma\rangle$$

$$\rho \vdash \langle x, \sigma\rangle \to \langle \omega, \sigma\rangle \text{ if } \omega = \rho(x)$$

$$\rho \vdash \langle \mathtt{fn}_\pi\ x\ \mathtt{=>}\ e, \sigma\rangle \to \langle \mathsf{close}\ (\mathtt{fn}_\pi\ x\ \mathtt{=>}\ e)\ \mathsf{in}\ \rho, \sigma\rangle$$

$$\rho \vdash \langle \mathtt{fun}_\pi\ f\ x\ \mathtt{=>}\ e, \sigma\rangle \to \langle \mathsf{close}\ (\mathtt{fun}_\pi\ f\ x\ \mathtt{=>}\ e)\ \mathsf{in}\ \rho, \sigma\rangle$$

$$\frac{\rho \vdash \langle e_1, \sigma_1\rangle \to \langle \mathsf{close}\ (\mathtt{fn}_\pi\ x\ \mathtt{=>}\ e)\ \mathsf{in}\ \rho', \sigma_2\rangle, \quad \rho \vdash \langle e_2, \sigma_2\rangle \to \langle \omega_2, \sigma_3\rangle, \quad \rho'[x \mapsto \omega_2] \vdash \langle e, \sigma_3\rangle \to \langle \omega, \sigma_4\rangle}{\rho \vdash \langle (e_1\ e_2)^l, \sigma_1\rangle \to \langle \omega, \sigma_4\rangle}$$

$$\frac{\rho \vdash \langle e_1, \sigma_1\rangle \to \langle \mathsf{close}\ (\mathtt{fun}_\pi\ f\ x\ \mathtt{=>}\ e)\ \mathsf{in}\ \rho', \sigma_2\rangle, \quad \rho \vdash \langle e_2, \sigma_2\rangle \to \langle \omega_2, \sigma_3\rangle, \quad \rho'[f \mapsto \mathsf{close}\ (\mathtt{fun}_\pi\ f\ x\ \mathtt{=>}\ e)\ \mathsf{in}\ \rho'][x \mapsto \omega_2] \vdash \langle e, \sigma_3\rangle \to \langle \omega, \sigma_4\rangle}{\rho \vdash \langle (e_1\ e_2)^l, \sigma_1\rangle \to \langle \omega, \sigma_4\rangle}$$

$$\frac{\rho \vdash \langle e_1, \sigma_1\rangle \to \langle \omega_1, \sigma_2\rangle, \quad \rho \vdash \langle e_2, \sigma_2\rangle \to \langle \omega_2, \sigma_3\rangle}{\rho \vdash \langle e_1\ \mathtt{;}\ e_2, \sigma_1\rangle \to \langle \omega_2, \sigma_3\rangle}$$

$$\frac{\rho \vdash \langle e, \sigma_1\rangle \to \langle \omega, \sigma_2\rangle}{\rho \vdash \langle \mathtt{ref}_\varpi\ e, \sigma_1\rangle \to \langle \iota, \sigma_2[\iota \mapsto \omega]\rangle} \text{ where } \iota \text{ is the first unused location}$$

$$\frac{\rho \vdash \langle e, \sigma_1\rangle \to \langle \iota, \sigma_2\rangle}{\rho \vdash \langle \mathtt{!}e, \sigma_1\rangle \to \langle \omega, \sigma_2\rangle} \text{ where } \omega = \sigma_2(\iota)$$

$$\frac{\rho \vdash \langle e_1, \sigma_1\rangle \to \langle \iota, \sigma_2\rangle, \quad \rho \vdash \langle e_2, \sigma_2\rangle \to \langle \omega, \sigma_3\rangle}{\rho \vdash \langle e_1\ \mathtt{:=}\ e_2, \sigma_1\rangle \to \langle \mathtt{()}, \sigma_3[\iota \mapsto \omega]\rangle}$$

$$\frac{\rho \vdash \langle e_1, \sigma_1\rangle \to \langle \omega_1, \sigma_2\rangle, \quad \rho[x \mapsto \omega_1] \vdash \langle e_2, \sigma_2\rangle \to \langle \omega_2, \sigma_3\rangle}{\rho \vdash \langle \mathtt{let}\ x\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2, \sigma_1\rangle \to \langle \omega_2, \sigma_3\rangle}$$

$$\frac{\rho \vdash \langle e, \sigma_1\rangle \to \langle \mathtt{true}, \sigma_2\rangle, \quad \rho \vdash \langle e_1, \sigma_2\rangle \to \langle \omega, \sigma_3\rangle}{\rho \vdash \langle \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2, \sigma_1\rangle \to \langle \omega, \sigma_3\rangle}$$

$$\frac{\rho \vdash \langle e, \sigma_1\rangle \to \langle \mathtt{false}, \sigma_2\rangle, \quad \rho \vdash \langle e_2, \sigma_2\rangle \to \langle \omega, \sigma_3\rangle}{\rho \vdash \langle \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2, \sigma_1\rangle \to \langle \omega, \sigma_3\rangle}$$

Table 7. Operational semantics.

A Semantics

The semantics is specified as a big-step operational semantics with environments $\rho \in \mathsf{Env}$ and stores $\sigma \in \mathsf{Store}$. The language has static scope rules and we give it a traditional call-by-value semantics. The semantic domains are:

$$\begin{aligned} \iota &\in \mathsf{Loc} = \cdots && \text{(unspecified)} \\ \omega &\in \mathsf{Val} \\ \omega &::= c \mid \mathsf{close}\ (\mathtt{fn}_\pi\ x\ \mathtt{=>}\ e)\ \mathsf{in}\ \rho \mid \mathsf{close}\ (\mathtt{fun}_\pi\ f\ x\ \mathtt{=>}\ e)\ \mathsf{in}\ \rho \mid \iota \end{aligned}$$

$$
\begin{array}{l}
R, M \ \rhd\ \texttt{ref}_{\varpi}\ e : S_1 \to S_3\ \&\ W' \\
\quad \text{iff } R, M \ \rhd\ e : S_1 \to S_2\ \&\ W \wedge \{(m, (\varpi, m)) \mid m \in M\} \subseteq W' \wedge \\
\qquad \forall m \in M : S_2 \oplus ((\varpi, m), W) \sqsubseteq S_3
\end{array}
$$

$$
\begin{array}{l}
R, M \ \rhd\ \texttt{!}e : S_1 \to S_2\ \&\ W' \\
\quad \text{iff } R, M \ \rhd\ e : S_1 \to S_2\ \&\ W \wedge \forall (m, (\varpi, m_d)) \in W : S_2(\varpi, m_d) \sqsubseteq (W', \mathsf{M})
\end{array}
$$

$$
\begin{array}{l}
R, M \ \rhd\ e_1\ \texttt{:=}\ e_2 : S_1 \to S_4\ \&\ W \\
\quad \text{iff } R, M \ \rhd\ e_1 : S_1 \to S_2\ \&\ W_1 \wedge R, M \ \rhd\ e_2 : S_2 \to S_3\ \&\ W_2 \wedge \\
\qquad \{(m, d_{()}) \mid m \in M\} \subseteq W \wedge \\
\qquad \forall (m, (\varpi, m_d)) \in W_1 : (S_3 \ominus (\varpi, m_d)) \oplus ((\varpi, m_d), W_2) \sqsubseteq S_4
\end{array}
$$

Table 8. Dealing with reference counts.

$$
\begin{array}{rcl}
e & ::= & \cdots \mid \iota \\
\rho & \in & \mathsf{Env} = \mathsf{Var} \to_{\text{fin}} \mathsf{Val} \\
\sigma & \in & \mathsf{Store} = \mathsf{Loc} \to_{\text{fin}} \mathsf{Val}
\end{array}
$$

The set Loc of locations for references is left unspecified. The judgements of the semantics have the form

$$\rho \vdash \langle e, \sigma_1 \rangle \to \langle \omega, \sigma_2 \rangle$$

and are specified in Table 7; the clauses themselves should be fairly straightforward. (We should also note that the choice of big-step operational semantics is not crucial for the development.)

B Reference Counts

An obvious extension of the work performed here is to incorporate an abstract notion of reference count for dynamically created cells. In the manner of [27] we could change the definition of $\widehat{\mathsf{Store}}$ (in Table 1) to have

$$
\begin{array}{rcl}
S \in \widehat{\mathsf{Store}} & = & \mathsf{Cell} \to (\widehat{\mathsf{Val}} \times \widehat{\mathsf{Pop}}) \\
p \in \widehat{\mathsf{Pop}} & = & \{\mathsf{O}, \mathsf{I}, \mathsf{M}\}
\end{array}
$$

Here the new $\widehat{\mathsf{Pop}}$ component denotes how many concrete locations may simultaneously be described by the abstract reference cell: O means zero, I means at most one, and M means arbitrarily many (including zero and one).

This makes it possible for the analysis sometimes to overwrite (as opposed to always augment) the value of a cell that is created or assigned. For this we need a new operation for adding a reference:

$$S \oplus ((\varpi, m), W) = S[(\varpi, m) \mapsto (W'', p'')]$$

where

$$\begin{aligned}(W', p') &= S(\varpi, m)\\ (W'', p'') &= \begin{cases}(W \cup W', \mathsf{M}) & \text{if } p' \neq \mathsf{O}\\ (W, \mathsf{I}) & \text{if } p' = \mathsf{O}\end{cases}\end{aligned}$$

We also need a new operation for removing a reference:

$$S \ominus (\varpi, m) = S[(\varpi, m) \mapsto (W'', p'')]$$

where

$$\begin{aligned}(W', p') &= S(\varpi, m)\\ (W'', p'') &= \begin{cases}(W', p') & \text{if } p' = \mathsf{M}\\ (\emptyset, \mathsf{O}) & \text{if } p' \neq \mathsf{M}\end{cases}\end{aligned}$$

The necessary modifications to the analysis are shown in Table 8.

A Per Model of Secure Information Flow in Sequential Programs

Andrei Sabelfeld and David Sands

Department of Computer Science
Chalmers University of Technology and the University of Göteborg
412 96 Göteborg, Sweden
{andrei,dave}@cs.chalmers.se

Abstract. This paper proposes an extensional semantics-based formal specification of secure information-flow properties in sequential programs based on representing degrees of security by partial equivalence relations (pers). The specification clarifies and unifies a number of specific correctness arguments in the literature, and connections to other forms of program analysis. The approach is inspired by (and equivalent to) the use of partial equivalence relations in specifying binding-time analysis, and is thus able to specify security properties of higher-order functions and "partially confidential data". We extend the approach to handle nondeterminism by using powerdomain semantics and show how *probabilistic security properties* can be formalised by using probabilistic powerdomain semantics.

1 Introduction

1.1 Motivation

You have received a program from an untrusted source. Let us call it company M. M promises to help you to optimise your personal financial investments, information about which you have stored in a database on your home computer. The software is free (for a limited time), under the condition that you permit a log-file containing a summary of your usage of the program to be automatically emailed back to the developers of the program (who claim they wish to determine the most commonly used features of their tool). Is such a program safe to use? The program must be allowed access to your personal investment information, and is allowed to send information, via the log-file, back to M. But how can you be sure that M is not obtaining your sensitive private financial information by cunningly encoding it in the contents of the innocent-looking log-file? This is an example of the problem of determining that the program has *secure information flow*. Information about your sensitive "high-security" data should not be able to propagate to the "low-security" output (the log-file). Traditional methods of access control are of limited use here since the program has legitimate access to the database.

This paper proposes an extensional semantics-based formal specification of secure information-flow properties in sequential programs based on representing

S.D. Swierstra (Ed.): ESOP/ETAPS'99, LNCS 1576, pp. 40–58, 1999.
© Springer-Verlag Berlin Heidelberg 1999

degrees of security by partial equivalence relations (pers[1]). The specification clarifies and unifies a number of specific correctness arguments in the literature, and connections to other forms of program analysis. The approach is inspired by and, in the deterministic case, equivalent to the use of partial equivalence relations in specifying binding-time analysis [HS91], and is thus able to specify security properties of higher-order functions and "partially confidential data" (e.g. one's financial database could be deemed to be partially confidential if the number of entries is not deemed to be confidential even though the entries themselves are). We show how the approach can be extended to handle nondeterminism, and illustrate how the various choices of powerdomain semantics affects the kinds of security properties that can be expressed, ranging from termination-insensitive properties (corresponding to the use of the Hoare (partial correctness) powerdomain) to *probabilistic security properties*, obtained when one uses a probabilistic powerdomain.

1.2 Background

The study of information flow in the context of systems with multiple levels of confidentiality was pioneered by Denning [Den76,DD77] in an extension of Bell and LaPadula's early work [BL76]. Denning's approach is to apply a static analysis suitable for inclusion into a compiler. The basic idea is that security levels are represented as a lattice (for example the two point lattice $PublicDomain \leq TopSecret$). The aim of the static analysis is to ensure that information from inputs, variables or processes of a given security level only flows to outputs, variables or processes which have been assigned a higher or equal security level.

1.3 Semantic Foundations of Information Flow Analysis

In order to verify a program analysis or a specific proof a program's security one must have a formal specification of what constitutes secure information flow. The value of a semantics-based specification for secure information flow is that it contributes significantly to the reliability of and the confidence in such activities, and can be used in the systematic design of such analyses. Many approaches to Denning-style analyses (including the original articles) contain a fair degree of formalism but arguably are lacking a rigorous soundness proof. Volpano *et al* [VSI96] claim to give the first satisfactory treatment of soundness of Denning's analysis. Such a claim rests on the dissatisfaction with soundness arguments based on an instrumented operational e.g., [Ørb95] or denotational semantics e.g., [MS92], or on "axiomatic" approaches which define security in terms of a program logic [AR80] without any models to relate the logic to the semantics of the programming language. The problem here is that an "instrumented semantics" or a "security logic" is just a definition, not subject to any further

[1] A Partial Equivalence relation is symmetric and transitive but not necessarily reflexive

mathematical justification. McLean points out [McL90] in a related discussion about the (non language-specific) Bell and LaPadula model:

> One problem is that ... they *[the Bell LaPadula security properties]* constitute a possible implementation of security, ... , rather than an abstract specification of what all secure systems must satisfy. By concerning themselves with particular controls over files inside the computer, rather than limiting themselves to the relation between input and output, they make it harder to reason about the requirements, ...

This criticism points to more abstract, *extensional* notions of soundness, based on, for example, the idea of *noninterference* introduced in [GM82].

1.4 Semantics-based Models of Information Flow

The problem of secure information flow, or "noninterference" is now quite mature, and very many specifications exist in the literature – see [McL94] for a tutorial overview. Many approaches have been phrased in terms of abstract, and sometimes rather ad hoc models of computation. Only more recently have attempts been made to rephrase and compare various security conditions in terms of well-known semantic models, e.g. the use of labelled transition systems and bisimulation semantics in [FG94]. In this paper we consider the problem of information-flow properties of sequential systems, and use the framework of *denotational semantics* as our formal model of computation. Along the way we consider some relations to specific static analyses, such as the ***S**ecurity **Lam**bda Calculus* [HR98] and an alternative semantic condition for secure information flow proposed by Leino and Joshi [LJ98].

1.5 Overview

The rest of the paper is organised as follows. **Section 2** shows how the per-based condition for soundness of binding times analysis is also a model of secure information flow. We show how this provides insight into the treatment of higher-order functions and structured data. **Section 3** shows how the approach can be adapted to the setting of a nondeterministic imperative language by appropriate use of a powerdomain-based semantics. We show how the choice of powerdomain (upper, lower or convex) affects the nature of the security condition. **Section 4** focuses on an alternative semantic specification due to Leino and Joshi. Modulo some technicalities we show that Leino's condition – and a family of similar conditions – are in agreement with, and can be represented using our form of specification. **Section 5** considers the problem of preventing unwanted *probabilistic* information flows in programs. We show how this can be solved in the same framework by utilising a probabilistic semantics based on the probabilistic powerdomain [JP89]. **Section 6** concludes.

2 A Per Model of Information Flow

In this section we introduce the way that *partial equivalence relations* (pers) can be used to model dependencies in programs. The basic idea comes from Hunts use of pers to model and construct abstract interpretations for strictness properties in higher-order functional programs [Hun90,Hun91], and in particular its use to model dependencies in *binding-time* analysis [HS91]. Related ideas already occur in the denotational formulation of live-variable analysis [Nie90].

2.1 Binding Time Analysis as Dependency Analysis

Given a description of the parameters in a program that will be known at partial evaluation time (called the *static* arguments), a binding-time analysis (BTA) must determine which parts of the program are dependent solely on these known parts (and therefore also known at partial evaluation time). The safety condition for binding time analysis must ensure that there is no dependency between the *dynamic* (i.e., non-static) arguments and the parts of the program that are deemed to be static. Viewed in this way, binding time analysis is purely an analysis of dependencies.[2]

Dependencies in Security In the security field, the property of absence of unwanted dependencies is often called *noninterference*, after [GM82]. Many problems in security come down to forms of dependency analysis. For example, in the case of *confidentiality*, the aim is to show that the outputs of a program which are deemed to be of low confidentiality do not have any dependence on inputs of a higher degree of confidentiality. In the case of *integrity* (*trust*), one must ensure that the value of some trusted data does not depend on some untrusted source.

Some intuitions about information flow Let us consider a program modelled as a function from some input domain to an output domain. Now consider the following simple functions mapping inputs to outputs: $\mathsf{snd} : D \times E \to E$ for some sets (or domains) D and E, and shift and test, functions in $\mathbf{N} \times \mathbf{N} \to \mathbf{N} \times \mathbf{N}$ and $\mathbf{N} \times \mathbf{N} \to \mathbf{N}$, defined by

$$\begin{aligned} \mathsf{snd}(x, y) &= y \\ \mathsf{shift}(x, y) &= (x + y, y) \\ \mathsf{test}(x, y) &= \mathsf{if}\ x > 0\ \mathsf{then}\ y\ \mathsf{else}\ y + 1 \end{aligned}$$

Now suppose that (h, l) is a pair where h is some high security information, and l is low, "public domain", information. Without knowing about what the actual values h and l might be, we know about the result of applying function snd will be a low value, and, in the case that we have a pair of numbers, the result

[2] Unfortunately, from the perspective of a partial evaluator, BTA is not *purely* a matter of dependencies; in [HS95] it was shown that the pure dependency models of [Lau89] and [HS91] are not adequate to ensure the safety of partial evaluation.

of applying shift will be a pair with a high first component and a low second component.

Note that the function test does not enjoy the same security property that snd does, since although it produces a value which is constructed from purely low-security components, the actual value is dependent on the first component of the input. This is what is known as an indirect information flow [Den76].

It is rather natural to think of these properties as "security types":

$$\begin{aligned} \mathsf{snd} &: \mathit{high} \times \mathit{low} \to \mathit{low} \\ \mathsf{shift} &: \mathit{high} \times \mathit{low} \to \mathit{high} \times \mathit{low} \\ \mathsf{test} &: \mathit{high} \times \mathit{low} \to \mathit{high} \end{aligned}$$

But what notion of "type", and what interpretation of "high" and "low" can formalise these more intuitive type statements? Interpreting types as sets of values is not adequate to model "high" and "low". To track degrees of dependence between inputs and outputs we need a more dynamic view of a type as a degree of variation. We must vary (parts of) the input and observe which (parts of) the output vary. For the application to confidentiality we want to determine if there is possible information leakage from a high level input to the parts of an output which are intended to be visible to a low security observer. We can detect this by observing whether the "low" parts of the output vary in any way as we vary the high input.

The simple properties of the functions snd and shift described above can be be captured formally by the following formulae:

$$\forall x, x', y.\, \mathsf{snd}(x, y) = \mathsf{snd}(x', y) \tag{1}$$

$$\forall x, x', y.\, \mathsf{snd}(\mathsf{shift}(x, y)) = \mathsf{snd}(\mathsf{shift}(x', y)) \tag{2}$$

Indeed, this kind of formula forms the core of the correctness arguments for the security analyses proposed by e.g., Volpano and Smith et al [VSI96,SV98], and also for the extensional correctness proofs in core of the *Slam-calculus* [HR98].

High and Low as Equivalence Relations We show how we can interpret "security types" in general as partial equivalence relations. We will interpret *high* (for values in D) as the equivalence relation All_D, and *low* as the relation Id_D where for all $x, x' \in D$:

$$x\ All_D\ x' \tag{3}$$

$$x\ Id_D\ x' \iff x = x'. \tag{4}$$

For a function $f : D \to E$ and binary relations $P \in Rel(D)$ and $Q \in Rel(E)$, we write $f : P \twoheadrightarrow Q$ iff

$$\forall x, x' \in D.\, x\ P\ x' \Longrightarrow (f\ x)\ Q\ (f\ x').$$

For binary relations P, Q we define the relation $P \times Q$ by:

$$(x, y)\ P \times Q\ (x', y') \iff x\ P\ x'\ \ \&\ \ y\ Q\ y'.$$

Now the security property of snd described by (1) can be captured by

$$\mathsf{snd} : All_D \times Id_E \multimap Id_E,$$

and (2) is given by

$$\mathsf{shift} : All_{\mathbf{N}} \times Id_{\mathbf{N}} \multimap All_{\mathbf{N}} \times Id_{\mathbf{N}}$$

2.2 From Equivalence Relations to Pers

We have seen how the equivalence relations *All* and *Id* may be used to describe security "properties" *high* and *low*. It turns out that these are exactly the same as the interpretations given to the notions "dynamic" and "static" given in [HS91]. This means that the binding-time analysis for a higher-order functional language can also be read as a security information-flow analysis. This connection between security and binding time analysis is already folk-law (See e.g. [TK97] for a comparison of a particular security type system and a particular binding-time analysis, and [DRH95] which shows how the incorporation of indirect information flows from Dennings security analysis can improve binding time analyses).

It is worth highlighting a few of the pertinent ideas from [HS91]. Beginning with the equivalence relations *All* and *Id* to describe *high* and *low* respectively, there are two important extensions to the basic idea in order to handle *structured data types* and *higher-order functions*. Both of these ideas are handled by the analysis of [HS91] which rather straightforwardly extends Launchbury's projection-based binding-time analysis [Lau89] to higher types. To some extent [HS91] anticipates the treatment of partially-secure data types in the SLam calculus [HR98], and the use of logical relations in their proof of noninterference.

For structured data it is useful to have more refined notions of security than just *high* and *low*; we would like to be able to model various *degrees* of security. For example, we may have a list of records containing name-password pairs. Assuming passwords are considered *high*, we might like to express the fact that although the whole list cannot be considered *low*, it can be considered as a (*low* × *high*)*list*. Constructing equivalence relations which represent such properties is straightforward – see [HS91] for examples (which are adapted directly from Launchbury's work), and [Hun91] for a more general treatment of finite lattices of "binding times" for recursive types.

To represent security properties of higher-order functions we use a less restricted class of relations than the equivalence relations. A *partial equivalence relation* (per) on a set D is a binary relation on D which is *symmetric* and *transitive*. If P is such a per let $|P|$ denote the domain of P, given by

$$|P| = \{x \in D \mid x\ P\ x\}\,.$$

Note that the domain and range of a per P are both equal to $|P|$ (so for any $x, y \in D$, if $x\ P\ y$ then $x\ P\ x$ and $y\ P\ y$), and that the restriction of P to $|P|$ is an equivalence relation. Clearly, an equivalence relation is just a per which is reflexive (so $|P| = D$). Partial equivalence relations over various applicative

structures have been used to construct models of the polymorphic lambda calculus (see, for example, [AP90]). As far as we are aware, the first use of pers in static program analysis is that presented in [Hun90].

For a given set D let $Per(D)$ denote the partial equivalence relations over D. $Per(D)$ is a meet semi-lattice, with meets given by set-intersection, and top element All.

Given pers $P \in Per(D)$ and $Q \in Per(E)$, we may construct a new per $(D \multimap E) \in Per(D \to E)$ defined by:

$$\begin{array}{c} f\ (P \multimap Q)\ g \\ \iff \\ \forall x, x' \in D.\, x\ P\ x' \Longrightarrow (f\ x)\ Q\ (g\ x'). \end{array}$$

If P is a per, we will write $x : P$ to mean $x \in |P|$. This notation and the above definition of $P \multimap Q$ are consistent with the notation used previously, since now

$$\begin{array}{rl} f : P \multimap Q & \iff f\ (P \multimap Q)\ f \\ & \iff \forall x, x' \in D.\, x\ P\ x' \Longrightarrow (f\ x)\ Q\ (f\ x'). \end{array}$$

Note that even if P and Q are both total (i.e., equivalence relations), $P \multimap Q$ may be partial. A simple example is $All \multimap Id$. If $f : All \multimap Id$ then we know that given a *high* input, f returns a *low* output. A constant function $\lambda x.42$ has this property, but clearly not all functions satisfy this.

2.3 Observations on Strictness and Termination Properties

We are interested in the security properties of functions which are the denotations of programs (in a Scott-style denotational semantics), and so there are some termination issues which should address. The formulation of security properties given above is sensitive to termination. Consider, for example, the following function $\mathsf{f} : \mathbf{N}_\perp \to \mathbf{N}_\perp$

$$\mathsf{f} = \lambda x.\mathsf{if}\ x > 0\ \mathsf{then}\ x\ \mathsf{else}\ fx$$

Clearly, if the argument is high then the result must be high. Now consider the security properties of the function $\mathsf{g} \circ \mathsf{f}$ where g the constant function $\mathsf{g} = \lambda x.2$. We might like to consider that g has type $high \to low$. However, if function application is considered to be strict (as in ML) then g is not in $|All_{\mathbf{N}_\perp} \multimap Id_{\mathbf{N}_\perp}|$ since $\perp\ All_{\mathbf{N}_\perp}\ 1$ but $\mathsf{g}(\perp) \neq \mathsf{g}(1)$. Hence the function $\mathsf{g} \circ \mathsf{f}$ does *not* have security type $high \to low$ (in our semantic interpretation). This is correct, since on termination of an application of this function, the low observer will have learned that the value of the high argument was positive.

The specific security analysis of e.g. the first calculus of Smith and Volpano [SV98] is termination sensitive – and this is enforced by a rather sweeping measure: all "while"-loop conditions must be low and all "while"-loop bodies must be low commands.

On the other hand, the type system of the SLam calculus [HR98] is not termination sensitive in general. This is due to the fact that it is based on a

call-by-value semantics, and indeed the composition $\mathsf{g} \circ \mathsf{f}$ could be considered to have a security type corresponding to "*high* $\rightarrow$ *low*". The correctness proof for noninterference carefully avoids saying anything about nonterminating executions. What is perhaps worth noting here is that had they chosen a non-strict semantics for application then the *same* type-system would yield termination sensitive security properties! So we might say that lazy programs are intrinsically more secure than strict ones. This phenomenon is closely related to properties of parametrically polymorphic functions [Rey83][3]. From the type of a polymorphic function one can predict certain properties about its behaviour – the so-called "free theorems" of the type [Wad89]. However, in a strict language one must add an additional condition in order that the theorems hold: the functions must be *bottom-reflecting* ($f(a) = \bot \implies a = \bot$). The same side condition can be added to make the e.g. the type system of the Slam-calculus termination-sensitive.

To make this observation precise we introduce one further constructor for pers. If $R \in Per(D)$ then we will also let R denote the corresponding per on $D_\bot$ without explicit injection of elements from D into elements in $D_\bot$. We will write $R_\bot$ to denote the relation in $Per(D_\bot)$ which naturally extends R by $\bot \; R \; \bot$.

Now we can be more precise about the properties of g under a strict (call-by-value) interpretation: $\mathsf{g} : (All_{\mathbf{N}})_\bot \multimap Id_{\mathbf{N}_\bot}$, which expresses that g is a constant function, modulo strictness. More informatively we can say that that $\mathsf{g} : (All_{\mathbf{N}}) \multimap Id_{\mathbf{N}}$ which expresses that g is a non-bottom constant function.

It is straightforward to express per properties in a subtype system of compositional rules (although we don't claim that such a a system would be in any sense complete). Pleasantly, all the expected subtyping rules are sound when types are interpreted as pers and the subtyping relation is interpreted as subset inclusion of relations. For the abstract interpretation presented in [HS91] this has already been undertaken by e.g. Jensen [Jen92] and Hankin and Le Métayer [HL94].

3 Nondeterministic Information Flow

In this section we show how the per model of security can be extended to describe nondeterministic computations. We see nondeterminism as an important feature as it arises naturally when considering the semantics of a concurrent language (although the treatment of a concurrent language remains outside the scope of the present paper.)

In order to focus on the essence of the problem we consider a very simplified setting – the analysis of commands in some simple imperative language containing a nondeterministic choice operator. We assume that there is some discrete (i.e., unordered) domain $\mathbf{St}$ of states (which might be viewed as finite maps from variables to discrete values, or simply just a tuple of values).

[3] Not forgetting that the use of Pers in static analysis was inspired, in part, by Abadi and Plotkin's Per model of polymorphic types [AP90]

3.1 Secure Commands in a Deterministic Setting

In the deterministic setting we can take the denotation of a command C, written $[\![C]\!]$, to be a function in $[\mathbf{St}_\perp \to \mathbf{St}_\perp]$, where by $[D_\perp \to E_\perp]$ we mean the set of strict and continuous maps between domains $D_\perp$ and $E_\perp$. Note that we could equally well take the set of all (trivially continuous) functions in $\mathbf{St} \to \mathbf{St}_\perp$, which is isomorphic.

Now suppose that the state is just a simple partition into a high-security half and a low-security half, so the set of states is the product $\mathbf{St}_{high} \times \mathbf{St}_{low}$. Then we might define a command C to be secure if no information from the high part of the state can leak into the low part:

$$C \text{ is secure} \iff [\![C]\!] : (All \times Id)_\perp \twoheadrightarrow (All \times Id)_\perp \tag{5}$$

Which is equivalent to saying that $[\![C]\!] : (All \times Id) \twoheadrightarrow (All \times Id)_\perp$ since we only consider strict functions. Note that this does not imply that $[\![C]\!]$ terminates, but what it does imply is that the termination behaviour is not influenced by the values of the high part of the state. It is easy to see that the sequential composition of secure commands is a secure command, since firstly, the denotation of the sequential composition of commands is just the function-composition of denotations, and secondly, in general for functions $g : D \to E$ and $f : E \to F$, and pers $P \in Per(D)$, $Q \in Per(E)$ and $R \in Per(F)$ it is easy to verify the soundness of the inference rule:

$$\frac{g : P \twoheadrightarrow Q \qquad f : Q \twoheadrightarrow R}{f \circ g : P \twoheadrightarrow R}$$

3.2 Powerdomain Semantics for Nondeterminism

A standard approach to giving meaning to a nondeterministic language – for example Dijkstra's guarded command language – is to interpret a command as a mapping which yields a *set* of results. However, when defining an ordering on the results in order to obtain a domain, there is a tension between the internal order of $\mathbf{St}_\perp$ and the subset order of the powerset. This is resolved by considering a suitable *powerdomain* structure [Plo76,Smy78]. The powerdomains are built from a domain D by starting with the *finitely generated* (f.g.) subsets of $D_\perp$ (those non-empty subsets which are either finite, or contain $\perp$), and a preorder on these sets. Quotienting the f.g. sets using the associated equivalence relation yields the corresponding domain. We give each construction in turn, and give an idea about the corresponding *discrete powerdomain* $\mathcal{P}[\mathbf{St}_\perp]$.

- *Lower (Hoare) powerdomain* Let $u \preceq_{\mathsf{L}} v$ iff $\forall x \in u.\, \exists y \in v.\, x \sqsubseteq y$. In this case the induced discrete powerdomain $\mathcal{P}_{\mathsf{L}}[\mathbf{St}_\perp]$ is isomorphic to the powerset of $\mathbf{St}$ ordered by subset inclusion. This means that the domain $[\mathbf{St}_\perp \to \mathcal{P}_{\mathsf{L}}[\mathbf{St}_\perp]]$ is isomorphic to all subsets of $\mathbf{St} \times \mathbf{St}$ – i.e. the relational semantics.
- *Upper (Smyth) powerdomain* The upper ordering on f.g. sets u, v, is given by

$$u \preceq_{\mathsf{U}} v \iff \forall y \in v.\, \exists x \in u.\, x \sqsubseteq y.$$

Here the induced discrete powerdomain $\mathcal{P}_{\mathsf{U}}[\mathbf{St}_\perp]$ is isomorphic to the set of finite non-empty subsets of $\mathbf{St}$ together with $\mathbf{St}_\perp$ itself, ordered by superset inclusion.

– *Convex (Plotkin) powerdomain* Let $u \preceq_{\mathsf{C}} v$ iff $u \preceq_{\mathsf{U}} v$ and $u \preceq_{\mathsf{L}} v$. This is also known as the Egli-Milner ordering. The resulting powerdomain $\mathcal{P}_{\mathsf{C}}[\mathbf{St}_\perp]$ is isomorphic to the f.g. subsets of $\mathbf{St}_\perp$, ordered by:

$$A \sqsubseteq_{\mathsf{C}} B \iff \begin{array}{l}\text{either } \perp \notin A \ \ \& \ \ A = B,\\ \text{or } \perp \in A \ \ \& \ \ A \setminus \{\perp\} \subseteq B\end{array}$$

A few basic properties and definitions on powerdomains will be needed. For each powerdomain constructor $\mathcal{P}[-]$ define the order-preserving "unit" map $\eta_D : D_\perp \to \mathcal{P}[D_\perp]$ which takes each element $a \in D$ into (the powerdomain equivalence class of) the singleton set $\{a\}$. For each function $f \in [D_\perp \to \mathcal{P}[E_\perp]]$ there exits a unique extension of f, denoted f^* where $f^* \in [\mathcal{P}[D_\perp] \to \mathcal{P}[E_\perp]]$ which is the unique mapping such that

$$f = f^* \circ \eta.$$

In the particular setting of the denotations of commands, it is worth noting that $[\![C_1; C_2]\!]$ would be given by:

$$[\![C_1; C_2]\!] = [\![C_2]\!]^* \circ [\![C_1]\!].$$

3.3 Pers on Powerdomains

Give one of the discrete powerdomains, $\mathcal{P}[\mathbf{St}_\perp]$, we will need a "logical" way to lift a per $P \in Per(\mathbf{St}_\perp)$ to a per in $Per(\mathcal{P}[\mathbf{St}_\perp])$.

Definition 1 *For each $R \in Per(D_\perp)$ and each choice of power domain $\mathcal{P}[-]$, let $\mathcal{P}[R]$ denote the relation on $\mathcal{P}[D_\perp]$ given by*

$$\begin{array}{rcl} A\ \mathcal{P}[R]\ B & \iff & \forall a \in A.\, \exists b \in B.\, a\ R\ b \\ & \& & \forall b \in B.\, \exists a \in A.\, a\ R\ b \end{array}$$

It is easy to check that $\mathcal{P}[R]$ is a per, and in particular that $\mathcal{P}[Id_{D_\perp}] = Id_{\mathcal{P}[D_\perp]}$.

Henceforth we shall restrict our attention to the semantics of simple commands, and hence the three discrete powerdomains $\mathcal{P}[\mathbf{St}_\perp]$.

Proposition 1 *For any $f \in [\mathbf{St}_\perp \to \mathcal{P}[\mathbf{St}_\perp]]$ and any R, $S \in Per(\mathbf{St}_\perp)$,*

$$f : R \multimap \mathcal{P}[S] \iff f^* : \mathcal{P}[R] \multimap \mathcal{P}[S]$$

From this it easily follows that the following inference rule is sound:

$$\frac{[\![C_1]\!] : P \multimap \mathcal{P}[Q] \qquad [\![C_2]\!] : Q \multimap \mathcal{P}[R]}{[\![C_1; C_2]\!] : P \multimap \mathcal{P}[R]}$$

3.4 The Security Condition

We will investigate the implications of the security condition under each of the powerdomain interpretations. Let us suppose that, as before the state is partitioned into a high part and a low part: $\mathbf{St} = \mathbf{St}_{high} \times \mathbf{St}_{low}$. With respect to a particular choice of powerdomain let the security "type" $C : high \times low \to high \times low$ denote the property

$$[\![C]\!] : (All \times Id)_\perp \to \mathcal{P}[(All \times Id)_\perp].$$

In this case we say that C is secure. Now we explore the implications of this definition on each of the possible choices of powerdomain:

1. In the lower powerdomain, the security condition describes in a weak sense termination-insensitive information flow. For example, the program

   $$\mathsf{if}\ h = 0\ \mathsf{then\ skip}\ [\!]\ \mathsf{loop\ else\ skip}$$

   (h is the high part of the state) is considered secure under this interpretation but the termination behaviours is influenced by h (it can fail to terminate only when $h = 0$).
2. In the upper powerdomain nontermination is considered catastrophic. This interpretation seems completely unsuitable for security unless one only considers programs which are "totally correct" – i.e. which must terminate on their intended domain. Otherwise, a possible nonterminating computation path will mask any other insecure behaviours a term might exhibit. This means that for *any* program C, the program $C\ [\!]\ \mathsf{loop}$ is secure!
3. The convex powerdomain gives the appropriate generalisation of the deterministic case in the sense that it is termination sensitive, and does not have the shortcomings of the upper powerdomain interpretation.

4 Relation to an Equational Characterisation

In this section we relate the Per-based security condition to a proposal by Leino and Joshi [LJ98]. Following their approach, assume for simplicity we have programs with just two variables: h and l of high and low secrecy respectively. Assume that the state is simple a pair, where h refers to the first projection and l is the second projection.

In [LJ98] the security condition for a program C is defined by

$$HH; C; HH = C; HH,$$

where "=" stands for semantic equality (the style of semantic specification is left unfixed), and HH is the program that "assigns to h arbitrary values" – aka "Havoc on H". We will refer to this equation as the equational security condition. Intuitively, the equation says that we cannot learn anything about the initial values of the high variables by variation of the low security variables.

The postfix occurrences of HH on each side mean that we are only interested in the final value of l. The prefix HH on the left-hand side means that the two programs are equal if the final value of l does not depend on the initial value of h.

In relating the equational security condition to pers we must first decide upon the denotation of HH. Here we run into some potential problems since it is necessary in [LJ98] that HH always terminates, but nevertheless exhibits unbounded nondeterminism. Although this appears to pose no problems in [LJ98] (in fact it goes without mention), to handle this we would need to work with non-ω-continuous semantics, and powerdomains for unbounded nondeterminism. Instead, we side-step the issue by assuming that the domain of h, $\mathbf{St}_{high}$, is finite.

4.1 Equational Security and Projection Analysis

A first observation is that the the equational security condition is strikingly similar to the well-known form of static analysis for functional programs known as projection analysis [WH87]. Given a function f, a projection analysis aims to find projections (continuous lower closure operators on the domain) α and β such that

$$\beta \circ f \circ \alpha = \beta \circ f$$

For (generalised) strictness analysis and dead-variable analysis, one is given β, and α is to be determined; for binding time analysis [Lau89] it is a forwards analysis problem: given α one must determine some β.

For strict functions (e.g., the denotations of commands) projection analysis is not so readily applicable. However, in the convex powerdomain HH is rather projection-like, since it effectively hides all information about the high variable; in fact it is an embedding (an upper closure operator) so the connection is rather close.

4.2 The Equational Security Condition Is Subsumed by the Per Security Condition

Hunt [Hun90] showed that projection properties of the form $\beta \circ f \circ \alpha = \beta \circ f$ could be expressed naturally as a per property of the form $f : R_\alpha \twoheadrightarrow R_\beta$ for equivalence relations derived from α and β by relating elements which get mapped to the same point by the corresponding projection.

Using the same idea we can show that the per-based security condition subsumes the equation specification in a similar manner.

We will establish the following:

Theorem 1. *For any command C*

$$[\![HH; C; HH]\!] = [\![C; HH]\!] \quad \textit{iff} \quad C : \mathit{high} \times \mathit{low} \to \mathit{high} \times \mathit{low}.$$

The idea will be to associate an equivalence relation to the function HH. More generally, for any command C let $ker(C)$, the *kernel* of C, denote the relation on $\mathcal{P}[\mathbf{St}_\perp]$ satisfying

$$s_1 \; ker(C) \; s_2 \iff [\![C]\!]s_1 = [\![C]\!]s_2.$$

Define the extension of $ker(C)$ by

$$A \; ker^*(C) \; B \iff [\![C]\!]^*A = [\![C]\!]^*B.$$

Recall the per interpretation of the type signature of C.

$$C : high \times low \to high \times low \iff [\![C]\!] : (All \times Id)_\perp \multimap \mathcal{P}[(All \times Id)_\perp].$$

Observe that $(All \times Id)_\perp = ker(HH)$ since for any h, l, h', l' it holds $[\![HH]\!](h, l) = [\![HH]\!](h', l')$ iff $l = l'$ iff $(h, l)(All \times Id)_\perp(h', l')$.

The proof of the theorem is based on this observation and on the following two facts:

- $\mathcal{P}[All \times Id]_\perp = ker^*(HH)$ and
- $[\![HH; C; HH]\!] = [\![C; HH]\!] \iff [\![C]\!] : ker(HH) \multimap ker^*(HH)$.

Let us first prove the latter fact by proving a more general statement similar to Proposition 3.1.5 from [Hun91] (the correspondence between projections and per-analysis). Note that we do not use the specifics of the convex powerdomain semantics here, so the proof is valid for any of the three choices of powerdomain.

Theorem 2. *Let us say that a command B is* idempotent *iff $[\![B; B]\!] = [\![B]\!]$. For any commands C and D, and any idempotent command B*

$$[\![B; C; D]\!] = [\![C; D]\!] \iff [\![C]\!] : ker(B) \multimap ker^*(D)$$

Corollary. Since $[\![HH]\!]$ is idempotent we can conclude that

$$[\![HH; C; HH]\!] = [\![C; HH]\!] \iff [\![C]\!] : ker(HH) \multimap ker^*(HH).$$

It remains to establish the first fact.

Theorem 3. $\mathcal{P}[All \times Id]_\perp = ker^*(HH)$

The proofs are given in the full version of the paper [SS99]. Thus, the equational and per security conditions in this simple case are equivalent.

In a more recent extension of the paper, [LJ99], Leino and Joshi update their relational semantics to handle termination-sensitive leakages and introduce abstract variables — a way to support partially confidential data. Abstract variables h and l are defined as functions of the concrete variables in a program. For example, for a list of low length and high elements l would be the length of the list and h would be the list itself. In the general case the choice of h and l could be independent, so an *independence condition* must be verified.

Abstract variables are easily represented in our setting. Suppose that some function $g \in \mathbf{St} \to D$ yields the value (in some domain D) of the abstract low variable from any given state, then we can represent the security condition on abstract variables by: $[\![C]\!] : R_g \multimap \mathcal{P}_\mathsf{C}[(All \times Id)_\perp]$ where $s_1 R_g s_2 \iff g\, s_1 = g\, s_2$.

5 A Probabilistic Security Condition

There are still some weaknesses in the security condition when interpreted in the convex powerdomain when it comes to the consideration of nondeterministic programs. In the usual terminology of information flow, we have considered *possibilistic information flows*. The probabilistic nature of an implementation may allow *probabilistic information flows* for "secure" programs. Consider the program

$$h := h \texttt{ mod } 100; (l := h \,[]\, l := \texttt{rand}(99)).$$

This program is secure in the convex powerdomain interpretation since regardless of the initial value of h, the final value of l can be any value in the range $\{0 \ldots 99\}$. But with a reasonably fair implementation of the nondeterministic choice and of the randomised assignment, it is clear that a few runs of the program, for a fixed input value of h, could yield a rather clear indiction of its value by observing only the possible final values of l, e.g., $2, 17, 2, 45, 2, 2, 33, 2, 97, 2, 8, 57, 2, 2, 66, \ldots$ from which we might reasonably conclude that the value of h was 2.

To counter this problem we consider *probabilistic powerdomains* [JP89] which allow the probabilistic nature of choice to be reflected in the semantics of programs, and hence enable us to capture the fact that varying the value of h causes *a change in the probability distribution* of values of l.

In the "possibilistic" setting we had the denotation of a command C to be a continuous function in $[\mathbf{St}_\perp \to \mathcal{P}_{\mathsf{C}}[\mathbf{St}_\perp]]$. In the probabilistic case, given an input to C not only we keep track of possible outputs, but also of probabilities at which they appear. Thus, we consider a domain $\mathcal{E}[\mathbf{St}_\perp]$ of distributions over $\mathbf{St}_\perp$. The denotation of C is going to be a function in $[\mathbf{St}_\perp \to \mathcal{E}[\mathbf{St}_\perp]]$.

The general probabilistic powerdomain construction from [JP89] on an inductive partial order $\mathcal{E}[D]$ is taken to be the domain of *evaluations*, which are certain continuous functions on $\Omega(D) \to [0, 1]$, where $\Omega(D)$ is the lattice of open subsets of D. We will omit a description of the general probabilistic powerdomain of evaluations since for the present paper it is sufficient and more intuitive to work with discrete domains, and hence a simplified notion of probabilistic powerdomain in terms of distributions.

If S is a set (e.g., the domain of states for a simple sequential language) then we define the probabilistic powerdomain of $S_\perp$, written $\mathcal{E}[S_\perp]$ to be the domain of *distributions* on $S_\perp$, where a distribution μ, to be a function from $S_\perp$ to $[0, 1]$ such that $\sum_{d \in S_\perp} \mu d = 1$. The ordering on $\mathcal{E}[S_\perp]$ is defined pointwise by $\mu \leq \nu$ iff $\forall d \neq \perp. \mu d \leq \nu d$. This structure is isomorphic to Jones and Plotkin's probabilistic powerdomain of evaluations for this special case.

As a simple instance of the probabilistic powerdomain construction from [JP89], one can easily see that $\mathcal{E}[S]$ is an inductively complete partial order with directed lubs defined pointwise, and with a least element $\omega = \eta_S(\perp)$, where η_S is the *point-mass distribution* defined for an $x \in S$ by

$$\eta_S(x)d = \begin{cases} 1, \text{ if } d = x, \\ 0, \text{ otherwise.} \end{cases}$$

To lift a function $f : D_1 \to \mathcal{E}[D_2]$ to type $\mathcal{E}[D_1] \to \mathcal{E}[D_2]$ we define the *extension* of f by

$$f^*(\mu)(y) = \sum_{x \in D_1} f(x)(y) * \mu(x).$$

The structure $(\mathcal{E}[D], \eta_D(x), *)$ is a *Kleisli triple*, and thus we have a canonical way of composing the probabilistic semantics of any two given programs. Suppose $f : D_1 \to \mathcal{E}[D_2]$ and $g : D_2 \to \mathcal{E}[D_3]$ are such. Then the lifted composition $(g^* \circ f)^*$ can be computed by one of the Kleisli triple laws as $g^* \circ f^*$.

The next step towards the security condition is to define how pers work on discrete probabilistic powerdomains. To lift pers to $\mathcal{E}[D]$ we need to consider a definition which takes into consideration the whole of each R-equivalence class in one go. The intuition is that an equivalence class of a per is a set of points that are indistinguishable by a low-level observer. For a given evaluation, the probability of a given observation by a low level user is thus the sum of probabilities over all elements of the equivalence class.

Define the per relation $\mathcal{E}[R]$ on $\mathcal{E}[D]$ for $\mu, \nu \in \mathcal{E}[D]$ by

$$\mu \; \mathcal{E}[R] \; \nu \text{ iff } \forall d \in |R|. \sum_{e \in [d]_R} \mu e = \sum_{e \in [d]_R} \nu e,$$

where $[d]_R$ stands for the $R-$equivalence class which contains d. Naturally, $\mu \; \mathcal{E}[Id] \; \nu \iff \mu = \nu$ and $\forall \mu, \nu \in \mathcal{E}[D]. \mu \; \mathcal{E}[All] \; \nu$.

As an example, consider $\mathcal{E}[(All \times Id)_\perp]$. Two distributions μ and ν in $(All \times Id)_\perp \to [0, 1]$ are equal if the probability of any given low value l in the left-hand distribution, given by $\sum_h \mu(h, l)$, is equal to the probability in the right-hand distribution, namely $\sum_h \nu(h, l)$.

The probabilistic security condition is indeed a strengthening of the possibilistic one – when we consider programs whose possibilistic and probabilistic semantics are in agreement.

Theorem 4. *Suppose we have a possibilistic (convex) semantics $[\![\cdot]\!]_{\mathsf{C}}$ and a probabilistic semantics $[\![\cdot]\!]_{\mathcal{E}}$, which satisfy a basic consistency property that for any command C, if $[\![C]\!]_{\mathcal{E}} i \, o > 0$ then $o \in [\![C]\!]_{\mathsf{C}} i$.*

Now suppose that R and S are equivalence relations on D. Suppose further that C is any command such that possibilistic behaviour agrees with its probabilistic behaviour, i.e., $o \in [\![C]\!]_{\mathsf{C}} i \implies [\![C]\!]_{\mathcal{E}} i \, o > 0$. Then we have that $[\![C]\!]_{\mathcal{E}} : R \twoheadrightarrow \mathcal{E}[S]$ implies $[\![C]\!]_{\mathsf{C}} : R \twoheadrightarrow \mathcal{P}_{\mathsf{C}}[S]$.

In the case that the state is modelled by a pair representing a high variable and a low variable respectively, it is easy to see that a command C is secure ($[\![C]\!]_{\mathcal{E}} : (All \times Id)_\perp \twoheadrightarrow \mathcal{E}[(All \times Id)_\perp]$) if and only if

$$\begin{gathered} [\![C]\!]_{\mathcal{E}}(i_h, i_l) \perp = [\![C]\!]_{\mathcal{E}}(i'_h, i_l) \perp \quad \text{and} \\ \textstyle\sum_{h \in \mathbf{St}_{high}} [\![C]\!]_{\mathcal{E}}(i_h, i_l)(h, o_l) = \sum_{h \in \mathbf{St}_{high}} [\![C]\!]_{\mathcal{E}}(i'_h, i_l)(h, o_l) \end{gathered}$$

for any i_l, i_h, i'_h and o_l. Intuitively the equation means that if you vary i_h the distribution of low variables (the sums provide "forgetting" the highs) does not change.

Let us introduce probabilistic powerdomain semantics definitions for some language constructs. Here we omit the $\mathcal{E}$-subscripts to mean the probabilistic semantics. Given two programs C_1, C_2 such that $[\![C_1]\!] : \mathbf{St}_\perp \to \mathcal{E}[\mathbf{St}_\perp]$ and $[\![C_2]\!] : \mathbf{St}_\perp \to \mathcal{E}[\mathbf{St}_\perp]$ the composition of two program semantics is defined by:

$$[\![C_1; C_2]\!]\ i\ o = \sum_{s \in \mathbf{St}_\perp} ([\![C_1]\!]\ i\ s) * ([\![C_2]\!]\ s\ o).$$

The semantics of the uniformly distributed nondeterministic choice $C_1 \,[\!]\, C_1$ is defined by $[\![C_1 \,[\!]\, C_2]\!]\ i\ o = 0.5[\![C_1]\!]\ i\ o + 0.5[\![C_2]\!]\ i\ o$. Consult [JP89] for an account of how to define the semantics of other language constructs.

Example. Recall the program

$$h := h \bmod 100; (l := h \,[\!]\, l := \mathsf{rand}(99))$$

Now we investigate the security condition by varying the value of h from 0 to 1. Take $i_l = 0, i_h = 0, i'_h = 1$ and $o_l = 0$. The left-hand side is

$$\sum_{h \in [0,\ldots,100]} [\![C]\!]\mathcal{E}(0,0)(h,0) = 0.5 * 1 + 0.5 * 0.01 = 0.505,$$

whereas the right-hand side is

$$\sum_{h \in [0,\ldots,100]} [\![C]\!]\mathcal{E}(1,0)(h,0) = 0.5 * 0 + 0.5 * 0.01 = 0.005.$$

So, the security condition does not hold and the program must be rejected.

Volpano and Smith recently devised a probabilistic security type-system [VS98] with a soundness proof based on a probabilistic operational semantics. Although the security condition that they use in their correctness argument is not directly comparable – due to the fact that they consider parallel deterministic threads and a non-compositional semantics – we can easily turn their examples into nondeterministic sequential programs with the same probabilistic behaviours. In the extended version of this paper [SS99] we show how their examples can all be verified using our security condition.

6 Conclusions

We have developed an extensional semantics-based specification of secure information flow in sequential programs, by embracing and extending earlier work on the use of partial equivalence relations to model binding times in [HS91]. We have shown how this idea can be extended to handle nondeterminism and also probabilistic information flow.

We recently became aware of work by Abadi, Banerjee, Heintze and Riecke [ABHR99] which shows that a single calculus (DCC), based on Moggi's computational lambda calculus, can capture a number of specific static analyses for

security, binding-time analysis, program slicing and call-tracking. Although their calculus does not handle nondeterministic language features, it is notable that the semantic model given to DCC is Per-based, and the logical presentations of the abstract interpretation for Per-based BTA from [HS91,Jen92,HL94] readily fit this framework (although this specific analysis is not one of those considered in [ABHR99]). They also show that what we have called "termination insensitive" analyses can be modelled by extending the semantic relations to relate bottom (nontermination) to every other domain point (without insisting on transitivity). It is encouraging to note that – at least in the deterministic setting – this appears to create no technical difficulties. We do not, however, see any obvious way to make the probabilistic security condition insensitive to termination in a similar manner.

We conclude by considering a few possible extensions and limitations:

Multi-level security There is no problem with handling lattices of security levels rather than the simple *high-low* distinction. But one cannot expect to assign any intrinsic semantic meaning to such lattices of security levels, since they represent a "social phenomenon" which is external to the programming language semantics. In the presence of multiple security levels one must simply formulate conditions for security by considering information flows between levels in a pairwise fashion (although of course a specific static analysis is able to do something much more efficient).

Downgrading and Trusting There are operations which are natural to consider but which cannot be modelled in an obvious way in an extensional framework. One such operation is the downgrading of information from high to low without losing information – for example representing the secure encryption of high level information. This seems impossible since an encryption operation does not lose information about a value and yet should have type $high \rightarrow low$ – but the only functions of type $high \rightarrow low$ are the constant functions. An analogous problem arises with Ørbæk and Palsberg's trust primitive if we try to use pers to model their *integrity analysis* [ØP97].

Concurrency Handling nondeterminism can be viewed as the main stepping stone to formulating a language-based security condition for concurrent languages, but this remains a topic for further work.

References

ABHR99. M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *POPL '99, Proceedings of the 26th Annual ACM Symposium on Principles of Programming Languages (January 1999)*, 1999.

AP90. M. Abadi and G. Plotkin. A per model of polymorphism and recursive types. In *Logic in Computer Science*. IEEE, 1990.

AR80. G. R. Andrews and R. P. Reitman. An axiomatic approach to information flow in programs. *ACM TOPLAS*, 2(1):56–75, January 1980.

BL76. D.E. Bell and L.J. LaPadula. Secure Computer Systems: Unified Exposition and Multics Interpretation. MTR-2997, Rev. 1, The MITRE Corporation, Bedford, Mass., 1976.

DD77. Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

Den76. Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

DRH95. M. Das, T. Reps, and P. Van Hentenryck. Semantic foundations of binding-time analysis for imperative programs. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 100–110, La Jolla, California, June 1995. ACM.

FG94. R. Focardi and R. Gorrieri. A classification of security properties for process algebra. *J. Computer Security*, 3(1):5–33, 1994.

GM82. Joseph Goguen and José Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, April 1982.

HL94. C. L. Hankin and D. Le Métayer. A type-based framework for program analysis. In *Proceedings of the First Static Analysis Symposium*, volume 864 of *LNCS*. Springer-Verlag, 1994.

HR98. Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Conference Record of POPL'98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 365–377, San Diego, California, January 19–21, 1998.

HS91. S. Hunt and D. Sands. Binding Time Analysis: A New PERspective. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'91)*, pages 154–164, September 1991. ACM SIGPLAN Notices 26(9).

HS95. F. Henglein and D. Sands. A semantic model of binding times for safe partial evaluation. In Manuel Hermenegildo and S.D̃oaitse Swierstra, editors, *Proc. Programming Languages: Implementations, Logics and Programs (PLILP), Utrecht, The Netherlands*, volume 982 of *Lecture Notes in Computer Science*, pages 299–320. Springer-Verlag, September 1995.

Hun90. S. Hunt. PERs generalise projections for strictness analysis. In *Draft Proceedings of the Third Glasgow Functional Programming Workshop*, Ullapool, 1990.

Hun91. L. S. Hunt. *Abstract Interpretation of Functional Languages: From Theory to Practice*. PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, 1991.

Jen92. T. P. Jensen. *Abstract Interpretation in Logical Form*. PhD thesis, Imperial College, University of London, November 1992. Available as DIKU Report 93/11 from DIKU, University of Copenhagen.

JP89. C. Jones and G. D. Plotkin. A probabilistic powerdomain of evaluations. In *Proceedings, Fourth Annual Symposium on Logic in Computer Science*, pages 186–195, Asilomar Conference Center, Pacific Grove, California, 5–8 June 1989. IEEE Computer Society Press.

Lau89. J. Launchbury. *Projection Factorisations in Partial Evaluation*. PhD thesis, Department of Computing, University of Glasgow, 1989.

LJ98. K. R. M. Leino and Rajeev Joshi. A semantic approach to secure information flow. In *MPC'98*, Springer Verlag LNCS, 1998.

LJ99. K. R. M. Leino and Rajeev Joshi. A semantic approach to secure information flow. *Science of Computer Programming*, 1999. To appear.

McL90. John McLean. The specification and modeling of computer security. *Computer*, 23(1):9–16, January 1990.

McL94. J. McLean. Security models. In J. Marciniak, editor, *Encyclopedia of Software Engineering*. Wiley & Sons, 1994.

MS92. M. Mizuno and D. Schmidt. A security flow control algorithm and its denotational semantics correctness proof. *Formal Aspects of Computing*, 4(6A):727–754, 1992.

Nie90. F. Nielson. Two-level semantics and abstract interpretation — fundamental studies. *Theoretical Computer Science*, (69):117–242, 90.

ØP97. Peter Ørbæk and Jens Palsberg. Trust in the λ-calculus. *Journal of Functional Programming*, 7(4), 1997.

Ørb95. Peter Ørbæk. Can you Trust your Data? In M. I. Schwartzbach P. D. Mosses and M. Nielsen, editors, *Proceedings of the TAPSOFT/FASE'95 Conference*, LNCS 915, pages 575–590, Aarhus, Denmark, May 1995. Springer-Verlag.

Plo76. G. D. Plotkin. A powerdomain construction. *SIAM Journal on Computing*, 5(3):452–487, 1976.

Rey83. John C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Proceedings 9th IFIP World Computer Congress, Information Processing '83, Paris, France, 19–23 Sept 1983*, pages 513–523. North-Holland, Amsterdam, 1983.

Smy78. Michael B. Smyth. Powerdomains. *Journal of Computer and Systems Sciences*, 16(1):23–36, February 1978.

SS99. Andrei Sabelfeld and David Sands. A per model of secure information flow in sequential programs. Technical report, Department of Computer Science, Chalmers University of Technology, 1999. http://www.cs.chalmers.se/~csreport/.

SV98. Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 355–364, 1998.

TK97. P. Thiemann and H. Klaeren. Binding-time analysis by security analysis. Universitt Tübingen, November 1997.

VS98. Dennis Volpano and Geoffrey Smith. Probabilistic noninterference in a concurrent language. In *11th IEEE Computer Security Foundations Workshop*, pages 34–43, 1998.

VSI96. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):1–21, 1996.

Wad89. Philip Wadler. Theorems for free. In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM, 1989.

WH87. P. Wadler and R. J. M. Hughes. Projections for strictness analysis. In *1987 Conference on Functional Programming and Computer Architecture*, pages 385–407, Portland, Oregon, September 1987.

Quotienting *Share* for Dependency Analysis

Andy King[1], Jan-Georg Smaus[1], and Pat Hill[2]

[1] University of Kent at Canterbury
Canterbury, CT2 7NF, UK
amk@ukc.ac.uk, jgs5@ukc.ac.uk
[2] School of Computer Studies
University of Leeds, Leeds, LS2 9JT, UK
hill@scs.leeds.ac.uk

Abstract. Def, the domain of definite Boolean functions, expresses (sure) dependencies between the program variables of, say, a constraint program. *Share*, on the other hand, captures the (possible) variable sharing between the variables of a logic program. The connection between these domains has been explored in the domain comparison and decomposition literature. We develop this link further and show how the *meet* (as well as the *join*) of Def can be modelled with efficient (quadratic) operations on *Share*. Further, we show how by compressing and widening *Share* and by rescheduling *meet* operations, we can construct a dependency analysis that is surprisingly fast and precise, and comes with time- and space- performance guarantees. Unlike some other approaches, our analysis can be coded straightforwardly in Prolog.
Keywords. (Constraint) logic programs, abstract interpretation, dataflow analysis, dependency analysis, definite Boolean functions, widening.

1 Introduction

Many analyses for logic programs, constraint logic programs and deductive databases use Boolean functions to express dependencies between program variables. In groundness analysis [2,4,10,20,26], the formula $x \wedge (y \leftarrow z)$ describes a state in which x is definitely ground, and there exists a grounding dependency such that whenever z becomes ground then so does y. Other useful properties like definiteness [5,21], strictness [19], and finiteness [6] can be also expressed and inferred with Boolean functions. Different classes of Boolean functions have different degrees of expressiveness. For example, Pos, the class of positive propositional formulae, has the condensing [1] property and is rich enough for goal-independent analysis. Def, the class of definite positive propositional formulae, is less expressive [1] but has been proposed for goal-dependent analysis of constraint programs [21].

The objective behind this work was to construct a goal-dependent groundness (and definiteness) analysis for logic (and constraint) programs, that was fast and precise enough to be practical, maintainable and easy to integrate into a Prolog compiler. Binary Decision Diagrams (BDD's) [7] (and their derivatives like ROBDD's) are the popular choice for implementing a dependency analysis [1,2,4,20,26]. These are essentially directed acyclic graphs in which identical sub-graphs are collapsed together. BDD operations require pointer manipulation and dynamic hashing [20] and thus BDD-based Pos analyses are usually

S.D. Swierstra (Ed.): ESOP/ETAPS'99, LNCS 1576, pp. 59–73, 1999.
© Springer-Verlag Berlin Heidelberg 1999

implemented in C [1,2,4,26]. Fecht [20] describes a notable exception that is coded in ML. The advantage of using ML is that it is more declarative than C and therefore easier to maintain. The disadvantage is that it impedes integration into a Prolog compiler [25]. The ideal, we believe, is to implement a dependency analyser in ISO Prolog. The problem, then, is essentially one of performance.

Our contribution to solving this problem is as follows: In terms of precision, we provide the first systematic precision experiments that compare *Pos* and *Def* for goal-dependent groundness (and definiteness) analysis. We found that *Def* was as precise as *Pos* for all our realistic Prolog and CLP($\mathcal{R}$) benchmarks. We build on this and demonstrate how *Def* can be implemented efficiently and coded succinctly in Prolog. Our starting point is the work of Cortesi *et al* [15,16] that shows that *Share*, which is a domain whose elements are sets of sets of variables, can be used to encode *Def*. We develop this to show:

- how the *meet* and *join* operations of *Def* can be computed straightforwardly based on this encoding, without the closure operation of *Share* [22] that has a worst-case exponential complexity;
- how an operation (that we call compression) aids fixpoint detection;
- how *meet* operations can be rescheduled to improve efficiency;
- how widening can be applied to ensure that both the time-complexity of the analysis (the number of iterations) and the space-complexity (the number of sets of variables), grows linearly in the size of the program;
- that the speed of our analysis compares surprisingly well against state-of-the-art BDD-based *Pos* analysers [4,20].

The rest of the paper is structured as follows. Section 2 surveys the necessary preliminaries. Section 3 recalls the relation between *Share* and *Def* and is included so that the paper is self-contained. Section 4 shows how the *meet* and *join* operations of *Def* can be computed efficiently using a *Share* based representation. Section 5 introduces compression and *meet* scheduling whereas Section 6 discusses widening. Section 7 describes the implementation. Section 8 reviews the related work, and finally Section 9 presents our conclusions.

2 Preliminaries

In this section, we introduce some notation and recall the definitions of Boolean functions and the domain *Share*. For a set S, $|S|$ denotes the cardinality and $\wp(S)$ the powerset of S. *Var* denotes a denumerable set (universe) of variables and $X \subset \mathit{Var}$ denotes a finite set of variables; the set of variables occurring in a syntactic object o is denoted by $\mathit{var}(o)$; the set of all idempotent substitutions is denoted by *Sub*; and *Bool* is defined to be $\{\mathit{true}, \mathit{false}\}$.

If $(S, \preceq)$ is a poset with top and bottom elements, and a *meet sqcap* and *join* $\sqcup$, then the 4-tuple $\langle S, \preceq, \sqcap, \sqcup\rangle$ denotes the corresponding lattice. A map $g : L \to K$, where L and K are lattices, is a homomorphism iff g is *join*-preserving and *meet*-preserving, that is, $g(a \sqcup b) = g(a) \sqcup g(b)$ and $g(a \sqcap b) = g(a) \sqcap g(b)$ for all $a, b \in L$. An isomorphism is a bijective homomorphism.

2.1 Boolean Functions

A Boolean function is a function $f : Bool^n \to Bool$ where $n \geq 0$. A Boolean function can be represented by a propositional formula over X where $|X| = n$. The set of propositional formulae over X is denoted by $Bool_X$. We use Boolean functions and propositional formulae interchangeably without worrying about the distinction [1]. We follow the convention of identifying a truth assignment with the set of variables that it maps to *true*.

Definition 1 ($model_X$). The (bijective) map $model_X : Bool_X \to \wp(\wp(X))$ is defined by: $model_X(f) = \{M \subseteq X \mid (\wedge M) \wedge (\neg \vee X \setminus M) \models f\}$. ∎

Example 1. If $X = \{x, y\}$, then the function $\{\langle true, true\rangle \mapsto true, \langle true, false\rangle \mapsto false, \langle false, true\rangle \mapsto false, \langle false, false\rangle \mapsto false\}$ can be represented by $x \wedge y$. Also $model_X(x \wedge y) = \{\{x, y\}\}$ and $model_X(x \vee y) = \{\{x\}, \{y\}, \{x, y\}\}$. ∎

Definition 2 (Pos_X, Def_X, Mon_X). Pos_X is the set of positive Boolean functions over X. A function f is positive iff $X \in model_X(f)$. Def_X is the set of positive functions over X that are definite. A function f is definite iff $M \cap M' \in model_X(f)$ for all $M, M' \in model_X(f)$. Mon_X is the set of monotonic Boolean functions over X. A function f is monotonic iff $M \in model_X(f)$ implies $M' \in model_X(f)$ for all M' such that $M \subseteq M' \subseteq X$. ∎

Note that $Def_X \subseteq Pos_X$ and $Mon_X \not\subseteq Pos_X$. It is possible to show that each $f \in Def_X$ is equivalent to a conjunction of definite (propositional) clauses, that is, $f = \wedge_{i=1}^{n}(y_i \leftarrow \wedge Y_i)$ [18].

Example 2. Suppose $X = \{x, y, z\}$ and consider the following table, which states, for some Boolean functions, whether they are in Def_X, Pos_X or Mon_X, and also gives $model_X$.

f	Def_X	Pos_X	Mon_X	$model_X(f)$
$false$			•	$\emptyset$
$x \wedge y$	•	•	•	$\{\{x, y\}, \{x, y, z\}\}$
$x \vee y$		•	•	$\{\{x\}, \{y\}, \{x, y\}, \{x, z\}, \{y, z\}, \{x, y, z\}\}$
$x \leftarrow y$	•	•		$\{\emptyset, \{x\}, \{z\}, \{x, y\}, \{x, z\}, \{x, y, z\}\}$
$x \vee (y \leftarrow z)$		•		$\{\emptyset, \{x\}, \{y\}, \{x, y\}, \{x, z\}, \{y, z\}, \{x, y, z\}\}$
$true$	•	•	•	$\{\emptyset, \{x\}, \{y\}, \{z\}, \{x, y\}, \{x, z\}, \{y, z\}, \{x, y, z\}\}$

Note, in particular, that $x \vee y$ is not in Def_X (since its set of models is not closed under intersection) and that $false$ is neither in Pos_X nor Def_X. ∎

Defining $f_1 \dot{\vee} f_2 = \wedge\{f \in Def_X \mid f_1 \models f \wedge f_2 \models f\}$, the 4-tuple $\langle Def_X, \models, \wedge, \dot{\vee}\rangle$ is a finite lattice [1], where *true* is the top element and $\wedge X$ is the bottom element. Existential quantification is defined by Schröder's Elimination Principle, that is, $\exists x.f = f[x \mapsto true] \vee f[x \mapsto false]$. Note that $\exists x.f \in Def_X$ if $f \in Def_X$ [1].

Example 3. If $X = \{x, y\}$ then $x \dot{\vee} (x \leftrightarrow y) = \wedge\{(x \leftarrow y), true\} = (x \leftarrow y)$, as can be seen in the Hasse diagram for Def_X (Fig. 1). Note also that $x \dot{\vee} y = \wedge\{true\} = true \neq (x \vee y)$. ∎

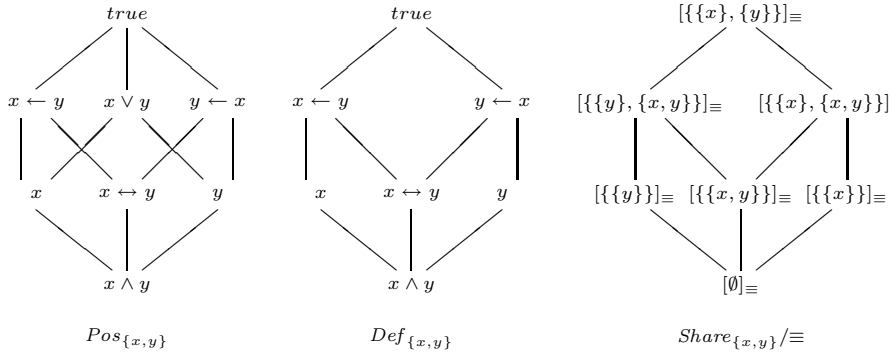


Fig. 1. Hasse diagrams

The maximum number of iterations of a fixpoint analysis relates to the length of the longest ascending chain in the underlying domain. For Pos_X, it is well-known that the longest chain has length $2^n - 1$ where $|X| = n$. It is less well-known that the same holds for Def_X.

Proposition 1. Let $|X| = n$. Let $f_1 \models f_2 \ldots \models f_k$ be a maximal strictly ascending chain where $f_i \in Def_X$ for all $i \in \{1, \ldots, k\}$. Then $k = 2^n$. ∎

2.2 Sharing Abstractions

For completeness, we introduce the basic ideas behind the *Share* domain [22]. This domain traces the possible variable sharing behaviour of a logic program. Two variables share if they are bound to terms that contain a common variable.

Definition 3 ($Share_X$)**.** $Share_X = \wp(\wp(X) \setminus \{\emptyset\})$. ∎

Thus we have the finite lattice $\langle Share_X, \subseteq, \cap, \cup\rangle$. The top element is $\wp(X) \setminus \{\emptyset\}$ and the bottom element is $\emptyset$.

Definition 4 (α^{sh}_X, γ^{sh}_X)**.** The abstraction map $\alpha^{sh}_X : \wp(Sub) \to Share_X$ is defined as $\alpha^{sh}_X(\Theta) = \{occ(\theta, v) \cap X \mid \theta \in \Theta \wedge v \in Var\} \setminus \{\emptyset\}$ where $occ(\theta, v) = \{x \in Var \mid v \in var(\theta(x))\}$. The concretisation map $\gamma^{sh}_X : Share_X \to \wp(Sub)$ is defined as $\gamma^{sh}_X(S) = \{\theta \in Sub \mid \alpha^{sh}_X(\{\theta\}) \subseteq S\}$. ∎

To streamline the theory and reduce the size of abstractions, the empty set is never included in a share set. However there is some loss of information. That is, if every element of Θ maps every element of X to a ground term then $\alpha^{sh}_X(\Theta) = \{\emptyset\} \setminus \{\emptyset\} = \emptyset = \alpha^{sh}_X(\emptyset)$. Thus α^{sh}_X (and hence γ^{sh}_X) cannot distinguish between a set of ground substitutions and the empty set. In practice, the empty set only arises when a computation fails and this would normally be flagged elsewhere in the analyser [9].

Example 4. Let $X = \{x, y, z\}$ and consider abstracting ⓐ $x = f(y, z)$ ⓑ where at program point ⓐ, no variable in X is ground or shares with any other element of X. The bindings on X, for example, could be θ_a or ϑ_a as given below. Then the bindings at ⓑ would be θ_b or ϑ_b, respectively.

$$\begin{array}{ll} \theta_a = \{y \mapsto g(u), z \mapsto v\} & \theta_b = \{x \mapsto f(g(u), v), y \mapsto g(u), z \mapsto v\} \\ \vartheta_a = \{x \mapsto f(u, u)\} & \vartheta_b = \{x \mapsto f(y, y), z \mapsto y\} \end{array}$$

The abstraction $S_a = \{\{x\}, \{y\}, \{z\}\}$ describes θ_a, that is $\theta_a \in \gamma_X^{sh}(S_a)$, since $occ(\theta_a, x) = \{x\}$, $occ(\theta_a, u) = \{u, y\}$, $occ(\theta_a, v) = \{v, z\}$ and $occ(\theta_a, y) = occ(\theta_a, z) = \emptyset$. Similarly $\vartheta_a \in \gamma_X^{sh}(S_a)$. The abstract unification operation of Jacobs and Langen [22] will compute the abstraction $S_b = \{\{x, y\}, \{x, z\}, \{x, y, z\}\}$ for the program point ⓑ. A safety result of Jacobs and Langen [22] asserts that $\theta_b, \vartheta_b \in \gamma_X^{sh}(S_b)$. Indeed, we see that $\theta_b \in \gamma_X^{sh}(S_b)$ since $occ(\theta_b, u) = \{u, x, y\}$, $occ(\theta_b, v) = \{v, x, z\}$, and $occ(\theta_b, x) = occ(\theta_b, y) = occ(\theta_b, z) = \emptyset$. The reader is encouraged to verify that $\vartheta_b \in \gamma_X^{sh}(S_b)$. ∎

3 Quotienting $Share_X$ to obtain Def_X

In this section we construct a homomorphism from $Share_X$ to Def_X. We recall the well-known connection between $Share_X$ and Def_X [13,14,15,16]. For the elements of $Share_X$, we define an abstraction α_X which interprets a sharing abstraction as representing a set of models and hence a Boolean function.

Definition 5 (α_X). The (abstraction) map $\alpha_X : Share_X \to Def_X$ is defined as follows: $\alpha_X(S) = model_X^{-1}(\{X \setminus (\cup S') \mid S' \subseteq S\})$. ∎

The definition of α_X is essentially that of α of Cortesi *et al* [14, Section 8.4], adapted to our definition of $Share_X$. α_X is well-defined, that is, $\alpha_X(S) \in Def_X$ for all $S \in Share_X$. First, since $X \in model_X(\alpha_X(S))$, it follows that $\alpha_X(S) \in Pos_X$. Secondly, if $M_1, M_2 \in model_X(\alpha_X(S))$ then $M_i = X \setminus (\cup S_i)$ where $S_i \subseteq S$ $(i = 1, 2)$. Clearly $S_1 \cup S_2 \subseteq S$. As $M_1 \cap M_2 = X \setminus (\cup(S_1 \cup S_2))$, it follows that $M_1 \cap M_2 \in model_X(\alpha_X(S))$.

Lemma 1. α_X is surjective. ∎

However, α_X is not injective, and thus it is a strict abstraction of $Share_X$. As an example, consider $X = \{x, y\}$, $S_1 = \{\{x\}, \{y\}\}$ and $S_2 = S_1 \cup \{\{x, y\}\}$. Then $\alpha_X(S_1) = {model_X}^{-1}(\{\emptyset, \{x\}, \{y\}, \{x, y\}\}) = \alpha_X(S_2)$ but $S_1 \neq S_2$.

Example 5. Let $X = \{x, y, z\}$ and $S = \{G_1, G_2, G_3\}$ where $G_1 = \{x\}$, $G_2 = \{y, z\}$ and $G_3 = \{z\}$. The table illustrates how $\alpha_X(S)$ can be computed by enumerating $\cup S'$ and $X \setminus (\cup S')$ for all $S' \subseteq S$.

S'	$\cup S'$	$X \setminus (\cup S')$	S'	$\cup S'$	$X \setminus (\cup S')$
$\emptyset$	$\emptyset$	$\{x, y, z\}$	$\{G_3\}$	$\{z\}$	$\{x, y\}$
$\{G_1\}$	$\{x\}$	$\{y, z\}$	$\{G_1, G_3\}$	$\{x, z\}$	$\{y\}$
$\{G_2\}$	$\{y, z\}$	$\{x\}$	$\{G_2, G_3\}$	$\{y, z\}$	$\{x\}$
$\{G_1, G_2\}$	$\{x, y, z\}$	$\emptyset$	$\{G_1, G_2, G_3\}$	$\{x, y, z\}$	$\emptyset$

Thus $\alpha_X(S) = {model_X}^{-1}(\{\emptyset, \{x\}, \{y\}, \{x, y\}, \{y, z\}, \{x, y, z\}\}) = (y \leftarrow z)$. The reader is encouraged to verify that $\alpha_X(\emptyset) = \wedge X$ and $\alpha_X(\{\{x\} \mid x \in X\}) = true$. ∎

It is perhaps easier to interpret an abstraction of $Share_X$ as definite Boolean functions by using the $\mathcal{C} : Share_X \to Def_X$ abstraction map of Cortesi *et al* [15,16]. $\mathcal{C}$ can be expressed particularly succinctly using the auxiliary operation *rel* which, given a set of variables G and an $S \in Share_X$, selects the subset of S which is relevant to the variables of G.

Definition 6 (*rel*). The map $rel : \wp(X) \times Share_X \to Share_X$ is defined by:

$$rel(Y, S) = \{G \in S \mid G \cap Y \neq \emptyset\}$$

Definition 7. The map $\mathcal{C} : Share_X \to Def_X$ is defined by $\mathcal{C}(S) = \wedge F$ where

$$F = \{y \leftarrow \wedge Y \mid y \in X \ \wedge \ Y \subseteq X \setminus \{y\} \ \wedge \ rel(\{y\}, S) \subseteq rel(Y, S)\}.$$

F is defined with $Y \subseteq X \setminus \{y\}$ rather than $Y \subseteq X$ to keep its size manageable.

Example 6. Consider again Example 5. The set of $Y \subseteq X \setminus \{x\}$ such that $rel(\{x\}, S) \subseteq rel(Y, S)$ is $\{\{x\}, \{x, y\}, \{x, z\}, \{x, y, z\}\}$. Likewise, set of $Y \subseteq X \setminus \{y\}$ such that $rel(\{y\}, S) \subseteq rel(Y, S)$ is $\{\{y\}, \{z\}, \{x, z\}, \{x, y\}, \{y, z\}, \{x, y, z\}\}$. Finally, the set of $Y \subseteq X \setminus \{z\}$ such that $rel(\{z\}, S) \subseteq rel(Y, S)$ is $\{\{z\}, \{x, z\}, \{y, z\}, \{x, y, z\}\}$. Thus $\mathcal{C}(S) = (y \leftarrow z)$.

The following proposition asserts the equivalence of $\mathcal{C}$ and α_X. It is proven by Cortesi *et al* [14], albeit for slightly different definitions. Modifying their proof to our definitions is straightforward.

Proposition 2. $\mathcal{C} = \alpha_X$.

By defining $S \equiv S'$ iff $\alpha_X(S) = \alpha_X(S')$, α_X induces an equivalence relation on $Share_X$ which quotients $Share_X$. Using the closure under union operation of Jacobs and Langen [22], we obtain a useful lemma about these equivalence classes.

Definition 8. *Let $S \in Share_X$. Then the closure under union $S^\star$ of S is defined by: $S^\star = \{\cup S' \mid S' \subseteq S\} \setminus \{\emptyset\}$.*

Note that closure under union is exponential.

Lemma 2. Let $S_1, S_2 \in Share_X$. Then ${S_1}^\star \equiv S_1$ and $S_1 \equiv S_2$ iff $S_1^\star = S_2^\star$

We lift α_X to $\alpha_X : Share_X/{\equiv} \to Def_X$ by defining $\alpha_X([S]_\equiv) = \alpha_X(S)$. Since $\alpha_X : Share_X \to Def_X$ is surjective it follows that $\alpha_X : Share_X/{\equiv} \to Def_X$ is bijective. We now define, for the the operations $\models$, $\dot\vee$ and $\wedge$ on Def_X, analogous operations $\sqsubseteq, \sqcup$ and $\sqcap$ on $Share_X/{\equiv}$.

Definition 9 ($\sqsubseteq$, $\sqcup$, $\sqcap$).

$$\begin{array}{rcl} [S_1]_\equiv \sqsubseteq [S_2]_\equiv & \leftrightarrow & \alpha_X([S_1]_\equiv) \models \alpha_X([S_2]_\equiv) \\ [S_1]_\equiv \sqcup [S_2]_\equiv & = & {\alpha_X}^{-1}(\alpha_X([S_1]_\equiv) \ \dot\vee \ \alpha_X([S_2]_\equiv)) \\ [S_1]_\equiv \sqcap [S_2]_\equiv & = & {\alpha_X}^{-1}(\alpha_X([S_1]_\equiv) \ \wedge \ \alpha_X([S_2]_\equiv)) \end{array}$$

Proposition 3. $\langle Share_X/{\equiv}, \sqsubseteq, \sqcap, \sqcup\rangle$ is a finite lattice.

It follows by construction that α_X is an isomorphism. For the dyadic case, the isomorphism is illustrated in Fig. 1.

4 Computing the *join* and *meet* within *Share*

In this section we show how the *meet* (as well as the *join*) of Def_X can be computed with $Share_X/\equiv$ via the isomorphism. It is not obvious from the definition of $\dot{\vee}$ how $f_1 \dot{\vee} f_2$ is computed, and it turns out that f_1 and f_2 must be put into (orthogonal) reduced monotonic body form [1]. In contrast, it is well-known [15,16] that with the *Share* representation, *join* basically reduces to set union.

Proposition 4. $[S_1]_\equiv \sqcup [S_2]_\equiv = [S_1 \cup S_2]_\equiv$ ∎

Example 7. Consider calculating $[S_1]_\equiv \sqcup [S_2]_\equiv$ where $X = \{w, x, y, z\}$, $S_1 = \{\{w, x, y\}, \{x, y\}, \{y\}, \{z\}\}$ and $S_2 = \{\{w, z\}, \{x\}, \{y\}, \{z\}\}$. Note that $\alpha_X(S_1) = (w \leftarrow x) \wedge (x \leftarrow y)$ and $\alpha_X(S_2) = w \leftarrow z$. Then $\alpha_X(S_1 \cup S_2) = \alpha_X(\{\{w, x, y\}, \{w, z\}, \{x\}, \{x, y\}, \{y\}, \{z\}\}) = (w \leftarrow (x \wedge z)) \wedge (w \leftarrow (y \wedge z))$ as required. ∎

The challenge is in defining a computationally efficient *meet*. This is defined in terms of a map *iff* which, in turn, is defined in terms of the binary-union operation of Jacobs and Langen [22]. We follow Cortesi *et al* [16] and denote binary union as $\otimes$.

Definition 10 (binary-union, $\otimes$). The map $\otimes : {Share_X}^2 \rightarrow Share_X$ is defined by: $S_1 \otimes S_2 = \{G_1 \cup G_2 \mid G_1 \in S_1 \wedge G_2 \in S_2\}$. ∎

The *if* and *iff* maps defined below are similar to the classical abstract unification operation of Jacobs and Langen [22]. Their interpretation, however, is that given variable sets Y_1 and Y_2 and an abstraction S such that $\alpha_X(S) = f$, *iff* and *if* compute new abstractions that represent $f \wedge (\wedge Y_1 \leftrightarrow \wedge Y_2)$ and $f \wedge (\wedge Y_1 \leftarrow \wedge Y_2)$.

Definition 11. The two maps $iff : \wp(X) \times \wp(X) \times Share_X \rightarrow Share_X$ and $if : \wp(X) \times \wp(X) \times Share_X \rightarrow Share_X$ are defined by: $iff(Y_1, Y_2, S) = (S \setminus (S_1 \cup S_2)) \cup (S_1 \otimes S_2)$ and $if(Y_1, Y_2, S) = (S \setminus S_1) \cup (S_1 \otimes S_2)$ where $S_1 = rel(Y_1, S)$ and $S_2 = rel(Y_2, S)$. ∎

One important difference between *iff* and *if* on the one hand and the abstract unification algorithm of Jacobs and Langen [22] on the other hand is that *iff* and *if* involve no costly closure calculations that arise because of the transitivity of variable sharing. Consequently the complexity *iff* and *if* is not exponential in the number of variable sets in S, but quadratic. This is a similar efficiency gain to that obtained with the *Share* pair-sharing quotient of Bagnara *et al* [3].

Proposition 5. $\alpha_X(iff(Y_1, Y_2, S)) = \alpha_X(S) \wedge (\wedge Y_1 \leftrightarrow \wedge Y_2)$ ∎

Corollary 1. $\alpha_X(if(Y_1, Y_2, S)) = \alpha_X(S) \wedge (\wedge Y_1 \leftarrow \wedge Y_2)$ ∎

Even though $if(Y_1, Y_2, S)$ can be simulated with $iff(Y_1', Y_2, S)$ where $Y_1' = Y_1 \cup Y_2$, it is cheaper to compute $rel(Y_1, S)$ than $rel(Y_1', S)$. This is one reason why $if(Y_1, Y_2, S)$ is more efficient than $iff(Y_1', Y_2, S)$. The map *if* is particularly useful in the analysis of constraint logic programs, where a constraint like $x = y + z$ is abstracted by $(x \leftarrow (y \wedge z)) \wedge (y \leftarrow (x \wedge z)) \wedge (z \leftarrow (x \wedge y))$.

Projection is an important component of a *Def* analysis within itself [21]. For completeness, we state its correctness as a proposition.

Definition 12 (projection $\overline{\exists}$). The map $\overline{\exists} : \wp(X) \times Share_X \to Share_X$ is defined by: $\overline{\exists}Y.S = \{G \cap Y \mid G \in S\} \setminus \{\emptyset\}$. ∎

Proposition 6. If $Y \subseteq X$ then $\exists(X \setminus Y).\alpha_X([S]_\equiv) = \alpha_Y([\overline{\exists}Y.S]_\equiv)$. ∎

Finally, Theorem 1 shows how *meet* can be computed with a sequence of *iff* calculations.

Theorem 1. $[S_1]_\equiv \sqcap [S_2]_\equiv = [\overline{\exists}X.S'_{n+1}]_\equiv$ where $X = \{x_1, \ldots, x_n\}$, $S'_1 = \rho(S_1) \cup S_2$, $S'_{j+1} = iff(\{\rho(x_j)\}, \{x_j\}, S'_j)$ for $j \in \{1, \ldots, n\}$ and ρ is a renaming such that $\rho(X) \cap X = \emptyset$. ∎

Note that $[S_1]_\equiv \sqcap [S_2]_\equiv$ could also be computed by $[S_1]_\equiv \sqcap [S_2]_\equiv = [{S_1}^\star \cap {S_2}^\star]_\equiv$. This, however, would be inefficient.

Example 8. Consider calculating $[S_1]_\equiv \sqcap [S_2]_\equiv$ where $X = \{w, x, y\}$, $S_1 = \{\{w, x\}, \{x\}, \{y\}\}$ and $S_2 = \{\{w\}, \{x, y\}, \{y\}\}$. Thus $\alpha_X(S_1) = w \leftarrow x$ and $\alpha_X(S_2) = x \leftarrow y$. If $\rho = \{w \mapsto w', x \mapsto x', y \mapsto y'\}$ then

$$\begin{array}{lll} S'_1 = & & \{\{w', x'\}, \{x'\}, \{y'\}, \{w\}, \{x, y\}, \{y\}\} \\ S'_2 = & iff(\{w'\}, \{w\}, S'_1) & = \{\{w', x', w\}, \{x'\}, \{y'\}, \{x, y\}, \{y\}\} \\ S'_3 = & iff(\{x'\}, \{x\}, S'_2) & = \{\{w', x', w, x, y\}, \{x', x, y\}, \{y'\}, \{y\}\} \\ S'_4 = & iff(\{y'\}, \{y\}, S'_3) & = \{\{w', x', y', w, x, y\}, \{x', y', x, y\}, \{y', y\}\} \end{array}$$

Thus $[S_1]_\equiv \sqcap [S_2]_\equiv = [\overline{\exists}X.{S_4}']_\equiv = \{\{w, x, y\}, \{x, y\}, \{y\}\}$. Observe that $\alpha_X([S_1]_\equiv \sqcap [S_2]_\equiv) = (w \leftarrow x) \wedge (x \leftarrow y)$ as required. ∎

5 Representing equivalence classes and *meet* rescheduling

In our analysis, the functions f and f' would be represented by elements of $Share_X$, S and S', say. The fixpoint stability check, $f = f'$, amounts to checking whether $[S]_\equiv = [S']_\equiv$ which, in turn, reduces to deciding whether $\alpha_X(S) = \alpha_X(S')$. To make this test efficient we represent an equivalence class by its smallest representative and thus introduce a compression operator c.

Definition 13. $c : Share_X \to Share_X$ is defined by: $c(S) = \cap\{S' \mid S' \equiv S\}$. ∎

The following proposition explains how $c(S)$ is actually computed.

Proposition 7. *Let* $n = |X|$. *Then* $c(S) = S_n$ *where* $S_1 = \{G \in S \mid |G| = 1\}$ *and* $S_{j+1} = S_j \cup \{G \in S \mid |G| = j + 1 \wedge G \notin S_j^\star\}$. ∎

Trivially, if $S \equiv S'$, then $c(S) = c(S')$. From the proposition we also see that $S^\star = {S_n}^\star = c(S)^\star$ and hence if $c(S) = c(S')$ then $S^\star = c(S)^\star = c(S')^\star = {S'}^\star$ so that $S \equiv S'$ by Lemma 2. Hence $c(S) = c(S')$ iff $S \equiv S'$ and thus by testing whether $c(S) = c(S')$ we can check for the fixpoint condition $S \equiv S'$.

When computing $c(S)$ we can test whether $G \notin {S_j}^\star$ without actually computing ${S_j}^\star$ as follows. Suppose $S_j = \{G_1, \ldots, G_m\}$ and ${G_0}' = G$. Then compute ${G_i}' = {G_{i-1}}' \setminus G_i$ if $G_i \subseteq G$ and put ${G_i}' = {G_{i-1}}'$ otherwise. Then ${G_m}' = \emptyset$ iff $G \in {S_j}^\star$. Using this tactic we can compute $c(S)$ in quadratic time.

Projection can sometimes lead to abstractions that include redundant variable sets as is illustrated below.

Example 9. Consider $S = \{\{x\},\{y\},\{x,y,z\}\}$ which, incidentally, represents $\alpha_X(S) = (z \leftarrow x) \wedge (z \leftarrow y)$. Projecting onto $\{x,y\}$ like so $\overline{\exists}\{x,y\}.S = \{\{x\},\{y\},\{x,y\}\}$ introduces the set $\{x,y\}$, whereas $c(\overline{\exists}\{x,y\}.S) = \{\{x\},\{y\}\}$. ∎

Compression is only applied to check for stability. In our framework, however, projection always precedes a stability check. For example, the answer pattern for a clause is obtained by projecting onto the head variables, and then a stability check is applied to see if other clauses need to be re-evaluated. Thus, in our framework, compression is applied after projection. Compression could be applied more widely though since, in general, $\mathit{iff}(Y_1,Y_2,c(S)) \neq c(\mathit{iff}(Y_1,Y_2,c(S)))$.

Example 10. Let $S = \{\{x\},\{y,x\},\{y,z\},\{x,z\}\}$. Then $c(S) = S$ and $\mathit{iff}(\{y\},\{z\},S) = \{\{x\},\{y,z\},\{y,x,z\}\}$ but $c(\{\{x\},\{y,z\},\{y,x,z\}\}) = \{\{x\},\{y,z\}\}$. ∎

In practice, however, the space saving yielded by $c(\mathit{iff}(Y_1,Y_2,S))$ over $\mathit{iff}(Y_1,Y_2,S)$ is usually small and not worth the effort of computing c.

Curiously, the efficiency of *meet* computations can often be significantly improved by introducing some redundancy into the representation. Specifically, a Boolean function is represented by a pair $\langle M,S\rangle$ where $M = X \setminus \mathit{var}(S)$. The pair $\langle M,S\rangle$ does not include any information that is not present in S: it simply flags those variables, M, that are ground (or definite). (This is reminiscent of the reactive ROBDD representation of Bagnara [2].) This is very useful in computing $[S_1]_{\equiv} \sqcap [S_2]_{\equiv}$ by the method prescribed in Theorem 1. Since *meet* is commutative, $[S_1]_{\equiv} \sqcap [S_2]_{\equiv}$ can be computed by the sequence $S'_1 = \rho(S_1) \cup S_2$, $S'_{j+1} = \mathit{iff}(\{\rho(x_{\pi(j)})\},\{x_{\pi(j)}\},S'_j)$ where π is a permutation on $\{1,\ldots,n\}$. The tactic is to choose a permutation with a maximal $m \in \{0,\ldots,n\}$ such that $(\rho(x_{\pi(1)}) \in M \vee x_{\pi(1)} \in M) \ldots (\rho(x_{\pi(m)}) \in M \vee x_{\pi(m)} \in M)$ where $M = M_1 \cup M_2$, $M_1 = X \setminus \mathit{var}(S_1)$ and $M_2 = X \setminus \mathit{var}(S_2)$. We call this technique *meet* rescheduling, and illustrate its usefulness in the following example.

Example 11. Consider $[S_1]_{\equiv} \sqcap [S_2]_{\equiv}$ where $X = \{x_1,x_2,x_3\}$, $S_1 = \{\{x_1,x_2\}\}$ and $S_2 = \{\{x_1\},\{x_2,x_3\}\}$. Thus $\alpha_X(S_1) = (x_1 \leftrightarrow x_2) \wedge x_3$ and $\alpha_X(S_2) = (x_2 \leftrightarrow x_3)$. Also $M_1 = \{x_3\}$, $M_2 = \emptyset$ and thus $M = \{x_3\}$. If $\rho = \{x_1 \mapsto x_1{}', x_2 \mapsto x_2{}', x_3 \mapsto x_3{}'\}$ then scheduling naively and using $\pi = \{1 \mapsto 3, 2 \mapsto 1, 3 \mapsto 2\}$ we obtain, respectively

$$\begin{array}{ll} S'_1 = \{\{x'_1,x'_2\},\{x_1\},\{x_2,x_3\}\} & S'_1 = \{\{x'_1,x'_2\},\{x_1\},\{x_2,x_3\}\} \\ S'_2 = \{\{x_1,x'_1,x'_2\},\{x_2,x_3\}\} & S'_2 = \{\{x'_1,x'_2\},\{x_1\}\} \\ S'_3 = \{\{x_1,x'_1,x'_2,x_2,x_3\}\} & S'_3 = \{\{x'_1,x_1,x'_2\}\} \\ S'_4 = \emptyset & S'_4 = \emptyset \end{array}$$

Note how the re-ordering π tends to reduce the size of the intermediate S'_i. ∎

A pair $\langle M,S\rangle$ representation is preferred to recomputing M prior to each *meet* because formulae typically occur as the operands of many *meet* operations. Thus M serves as a memo, avoiding unnecessary recomputation.

6 Widening

Apart from reducing the size of abstractions, it is also worthwhile to avoid generating large abstractions that can arise from the quadratic growth of $\mathit{iff}(Y_1, Y_2, S)$ and $\mathit{if}(Y_1, Y_2, S)$ stemming from $S_1 \otimes S_2$. However, if $|S| = n$, $|S_1| = n_1$, $|S_2| = n_2$ then $|\mathit{iff}(Y_1, Y_2, S)| \leq n + n_1 n_2 - (n_1 + n_2)$. Thus it is possible to detect that $|\mathit{iff}(Y_1, Y_2, S)|$ will definitely be small, say less than a threshold k, without computing $\mathit{iff}(Y_1, Y_2, S)$ itself. This leads to the following (widened) versions of *iff* and *if* that trade precision for efficiency.

Definition 14.

$$\mathit{iff}_k(Y_1, Y_2, S) = \begin{cases} \mathit{iff}(Y_1, Y_2, S) & \text{if } n+n_1n_2-(n_1+n_2) \leq k \\ S & \text{otherwise} \end{cases} \qquad \mathit{if}_k(Y_1, Y_2, S) = \begin{cases} \mathit{if}(Y_1, Y_2, S) & \text{if } n+n_1n_2-n_1 \leq k \\ S & \text{otherwise} \end{cases}$$

where $S_1 = rel(Y_1, S)$, $S_2 = rel(Y_2, S)$, $|S| = n$, $|S_1| = n_1$ *and* $|S_2| = n_2$. ∎

This (space) widening ensures that at each stage of the analysis the size of an abstraction is kept smaller than k. In fact, since the size of the abstraction depends on the number of variables, k is defined as a multiple of the number of the variables in a clause. This is enough to ensure that, in our interpreter, our space usage grows linearly with the size of the program. A widened *meet* can be obtained by replacing each $\mathit{iff}(\{\rho(x_j)\}, \{x_j\}, S'_j)$ of Theorem 1 by $\mathit{iff}_k(\{\rho(x_j)\}, \{x_j\}, S'_j)$. (Interestingly, a widening for ROBDD's is described by Fecht [20] that combats the space problems that arise in the analysis of high arity predicates.)

Folklore [8] says that call and answer patterns rarely get updated more than 3–4 times. This is true for many small programs, but in chat_80.pl and aqua_c.pl we have observed patterns being updated 10–12 times. To bound the number of iterations that can occur, we widen abstractions if they are updated more than, say, 8 times. This (time) widening is defined by: $\triangle(S) = S' \cup \{\{x\} \mid x \in var(S) \setminus var(S')\}$ where $S' = \{G' \in S \mid \forall G \in S.(G \cap G' \neq \emptyset) \rightarrow (G' \subseteq G)\}$. Observe that $[S]_{\equiv} \sqsubseteq [\triangle(S)]_{\equiv}$ and that $\alpha_X(\triangle(S)) = (\wedge Y) \wedge (\wedge\{x \leftrightarrow y \mid G \in \triangle(S) \wedge x, y \in G\})$ where $Y = X \setminus var(\triangle(S))$. Formulae of this form occur in the $WPos$ domain of Codish *et al* [11] and thus have a maximal chain length that is linear in $|X|$. This ensures that the number of iterates will be linear in the sum of the arities of program predicates, and thus provides a time guarantee for a cautious compiler vendor.

7 Experimental work

To investigate whether a quadratic *meet*, *meet* rescheduling and widening are enough to obtain an efficient and scalable dependency analysis, we have implemented an analyser in Prolog as a simple meta-interpreter that uses induced magic-sets [9] and eager evaluation [27] to perform goal-dependent bottom-up evaluation. Induced magic is a refinement of the magic set transformation, avoiding much of the re-computation that arises because of the repetition of literals in the bodies of magic'ed clauses [9]. It also avoids the overhead of applying the

magic set transformation. Eager evaluation [27] is a fixpoint iteration strategy which proceeds as follows: whenever an atom is updated with a new (less precise) abstraction, a recursive procedure is invoked to ensure that every clause that has that atom in its body is re-evaluated. Eager evaluation can involve more re-computation than semi-naive iteration but it has the advantages that (1) a (Δ-)set of recently updated atoms does not need to be represented; (2) eager evaluation performs a depth-first traversal of the call-graph so that information about strongly connected components (SCCs) of the call-graph is not as important as in semi-naive iteration. Thus we also avoid computing SCCs.

		fixpoint					precision			
file	abs	*Con*	Def_n	Def_r	Def_w	*Pos*	*Con*	*Def*	Def_w	*Pos*
disj_r.pl	0.13	0.06	0.13	0.09	0.08	0.03	38	97	97	97
scc1.pl	0.21	0.13	0.81	0.81	0.77	0.37	44	89	89	89
tictactoe.pl	0.22	0.02	0.12	0.08	0.09	0.04	56	56	56	56
dialog.pl	0.13	0.03	0.1	0.07	0.07	0.04	46	70	70	70
ime_v2-2-1.pl	0.18	0.05	0.56	0.3	0.29	0.36	77	101	101	101
cs_r.pl	0.26	0.06	0.11	0.1	0.09	0.05	36	149	149	149
flatten.pl	0.17	0.03	0.76	0.36	0.35	1.62	26	27	27	27
conman.pl	0.2	0.0	0.01	0.02	0.01	0.01	6	6	6	6
unify.pl	0.21	0.04	0.87	0.45	0.44	59.06	69	70	70	70
bridge.clpr	0.34	0.01	0.06	0.03	0.03	0.04	24	24	24	24
neural.clpr	0.25	0.05	0.58	0.15	0.14	0.12	80	118	118	118
kalah.pl	0.22	0.1	0.12	0.13	0.12	0.04	91	199	199	199
bryant.pl	0.28	0.06	1.46	1.04	1.03	70.24	89	89	89	89
nbody.pl	0.33	0.05	8.44	0.33	0.32	1.03	83	109	109	109
sdda.pl	0.25	0.04	0.34	0.38	0.38	3.33	17	17	17	17
peep.pl	0.51	0.04	0.37	0.3	0.29	0.87	8	10	10	10
boyer.pl	0.32	0.03	0.11	0.18	0.19	0.15	3	3	3	3
read.pl	0.38	0.05	0.81	0.42	0.4	1.15	90	99	99	99
qplan.pl	0.36	0.12	1.77	1.59	1.56	63.8	42	49	49	49
reducer.pl	0.31	0.04	∞	∞	0.93	∞	36	–	41	–
press.pl	0.36	0.17	11.6	4.23	1.26	2.29	45	53	53	53
asm.pl	0.51	0.07	0.33	0.39	0.43	0.23	86	87	87	87
parser_dcg.pl	0.39	0.08	0.53	0.58	0.55	0.97	25	41	41	41
trs.pl	0.52	0.09	3.08	2.51	2.48	∞	13	13	13	∞
dbqas.pl	0.36	0.02	0.31	0.09	0.09	0.23	36	43	43	43
ann.pl	0.41	0.09	1.9	1.27	0.94	1.99	73	73	73	73
nand.pl	0.49	0.62	0.67	0.39	0.39	0.16	123	402	402	402
simple_analyzer.pl	0.38	0.08	2.37	0.74	0.72	∞	88	89	89	–
sim.pl	0.76	0.18	3.62	2.51	2.46	∞	81	100	100	–
ili.pl	0.61	0.13	∞	∞	1.69	19.16	4	–	4	4
lnprolog.pl	0.41	0.16	0.37	0.53	0.5	0.23	54	143	143	143
rubik.pl	0.79	0.2	2.0	1.96	1.93	∞	153	160	160	–
strips.pl	0.79	0.04	0.17	0.16	0.16	0.06	144	144	144	144
peval.pl	0.68	0.08	1.92	1.49	1.06	9.34	27	27	27	27
sim_v5-2.pl	0.86	0.11	0.48	0.55	0.54	0.49	100	101	101	101
chat_parser.pl	1.09	0.62	4.27	3.88	3.52	∞	444	505	505	–
aircraft.pl	2.01	8.56	1.34	1.33	1.3	0.35	228	687	687	687
essln.pl	1.49	0.25	2.76	1.43	1.39	17.44	103	155	155	155
chat_80.pl	4.63	1.23	12.89	9.99	9.82	∞	457	839	839	–
aqua_c.pl	12.17	7.07	∞	∞	69.56	∞	1087	–	1227	–

The table summarises our experimental results for applying *Def* to some of the largest Prolog and CLP($\mathcal{R}$) benchmark programs that we could find on the WWW. The programs are ordered by size, where size is measured in terms of the number of (distinct abstract) clauses. To assess the precision of the *Def* analysis, we have implemented a standard *Pos* analysis following the technique of Codish and Demoen [10]. Ideally our *Def* analysis should match its precision. We have also modified this analysis to obtain a *Con* analysis [23]. Ideally our *Def* analysis

should significantly improve on its precision, since otherwise neither Def or Pos are worthwhile! For completeness, we have included the timings for Pos and Con, but we are primarily concerned with precision. Our Pos analysis is not state-of-the-art. The *abs* column give the time for parsing the files and abstracting them, that is, replacing built-ins, like `arg(X, T, S)`, with formulae, like $\mathtt{X} \wedge (\mathtt{S} \leftarrow \mathtt{T})$. This overhead is the same for all the analyses. The *fixpoint* columns gives the time to compute the fixpoint. Def_n is a naive implementation of our analysis (that took two person weeks to construct) which applies compression but not *meet* rescheduling and widening; Def_r additionally applies *meet* rescheduling; and Def_w applies compression, *meet* rescheduling and widening. The Def_r and Def_w analysers were developed together and took an additional 4 days to construct. The code for Def_n, Def_r and Def_w meta-interpreters (including all the set manipulation utilities) is less than 700 clauses. We widen for time at iteration 8 and widen for space when the number of variable sets is more than 16 times the number of variables in a clause. Times are in seconds and ∞ indicates that the fixpoint calculation timed out after two minutes. The timings were carried out on an Sun-20 SuperSparc with 64 MByte to match the architecture of Fecht [20]. The analysers were coded in SICStus 3#5 and compiled to naive code. The *precision* columns give the total number of ground arguments in the call and answer patterns: this is an absolute measure which reflects the usefulness of the analysis for code optimisation. The precision figures for Def_n and Def_r are the same and given in column Def.

The experimental results indicate that Def_w has good scaling behaviour. This is the crucial point. Put simply, there are no programs for which Pos terminates within two minutes and Def_w does not (although Pos is sometimes faster). Usually *meet* rescheduling gives a speedup and sometimes this speedup is very dramatic. 10% of the programs, however, run slower with *meet* rescheduling. This typically occurs in programs with very few ground arguments where the effort of rescheduling in not repaid by a reduction in the size of sharing abstractions. Widening seems to be crucial for scalability as is illustrated by reducer, ili and aqua_c. Widening, in fact, is rarely applied. It is crucial for efficiency though because, just one large sharing abstraction can have a disastrous impact on performance. (This also suggests that widening is necessary in the pair-sharing quotient of *Share* [3].)

Since our machine matches that of Fecht [20] we can also compare the speed of our Def analyser to the BDD-based Pos GENA analyser [20]. This the one of the fastest (perhaps the fastest) Pos analysis that is described in the literature. With the sophisticated CallWDFS [20] framework, ann.pl takes 0.18 s, nand.pl takes 0.31 s, chat_80.pl takes 4.29 s, and aqua_c.pl takes 28.54 s. Since Fecht [20] does not give processor details for his Sparc-20, we have run our experiments on the slowest 50MHz model that was manufactured. His machine could well be almost twice as fast. Even though our framework is not semi-naive, we are (at most) 2–4 times as slow as GENA. Furthermore, to perform a comparison against China instantiated with Pos [4], Bagnara has run Def_n and China on a Pentium 200MHz PC with 64 MByte of memory. On trs.pl and chat_80.pl Def_n

take 3.17 s and 12.59 s respectively running interpreted SICStus 3#6 bytecode. China takes 2.94 s and 6.24 s respectively. It seems reasonable to assume that with Def_w on the same PC, trs.pl and chat_80.pl would take $3.17 \times \frac{2.48}{3.08} \approx 2.55$ s and $12.59 \times \frac{9.82}{12.89} \approx 9.59$ s. This performance gap for chat_80.pl would be closed if naive code assembly was available for the PC. To summarise, the experimental results are very encouraging and despite the simplicity of the interpreter, our Def_w analysis appears to be fast, precise and scalable and, of course, can be implemented easily in Prolog.

8 Related work

Cortesi *et al* [15] first pointed out that *Share* expresses the groundness dependencies of Def. Quotienting was introduced by Cortesi *et al* [16] as a systematic way of obtaining the reference domain of [15]. Like Bagnara *et al* [3], we do not fully adhere to the quotienting terminology and methodology of Cortesi *et al* [15] but rather follow the standard convention [17] of inducing an equivalence relation ($\equiv$) from an abstraction map (α_X). Also, Lemma 6.2 of [16] can be interpreted as a way of computing the *meet* in *Def* with the classic abstract unification of Jacobs and Langen [22]. We take this further and show how the *meet* can be computed without exponential time closure operations.

Bagnara *et al* [3] point out that *Share* includes redundant sharing information with respect to pair-sharing. This work is related to ours in that our domain may be viewed as a further quotient of the pair-sharing domain. However, widening has not been explored for the pair-sharing domain although, we have shown that even for our simpler domain, that widening is crucial for scalability.

Armstrong *et al* [1] investigate various normal forms of Boolean functions and the relative precision of Pos and Def. C-based implementations of each representation are described. For the representations of Pos, it is concluded that ROBDD's give the fastest analysis. A specialised representation for Def, based on Dual Blake Canonical Form (DBCF), is found to be the fastest overall. For medium-sized programs it is several times faster than ROBDD's, and it is concluded that this is the representation likely to scale best for real programs. The precision achieved using Pos was found to be significantly higher than Def, although it is remarked that a top-down analyser would improve the precision of Def since it is not condensing. Our findings support this remark.

Bagnara and Schachte [4] develop the idea [2] that a hybrid implementation of ROBDD's that keeps definite information separate from dependency information is more efficient than keeping the two together. This hybrid representation can significantly decrease the size of ROBDD's and thus is a useful implementation tactic. A comparison with our *Def* analysis has already been given. Fecht [20] compares his *Pos* analyser to that of Van Hentenryck *et al* [26] and concludes that his analyser is an order of magnitude faster. For reasons of space, the reader is referred to [20, pp. 305–307] for more details. Performance figures for another hybrid representation are given in [24]. We just observe that [4] and [20] are very good systems to measure against.

García de la Banda *et al* [21] represent *Def* functions in terms of a domain $\wp(X) \times \wp(X \times \wp(\wp(X)))$, so that the Herbrand constraint $x = f(y, z)$, for example, is represented by $\langle \emptyset, \{\langle x, \{\{y, z\}\}\rangle, \langle y, \{\{x\}\}\rangle, \langle z, \{\{x\}\}\rangle\}\rangle$ which encodes $x \leftrightarrow (y \wedge z)$. Abstract conjunction is expressed in terms of six rewrite rules that put conjunctions of formulae into a normal form. Although not stated, the normal form is essentially the (orthogonal) reduced monotonic body form [1] in which a definite function is represented as $f = \wedge_{x \in X}(x \leftarrow M_x)$ where $M_x \in Mon_X$ and $x \notin M_x$. Orthogonality ensures that the *meet* is safe. Our work shows how this symbolic manipulation of definite function can be replaced with a simpler domain and simpler *join* and *meet* operations.

Corsini *et al* [12] describe how variants of *Pos* can be implemented using Toupie, a constraint language based on the μ-calculus. This BDD-based analysis appears to be at least five times as fast as [26] for success pattern analysis. Thus, if the analyser was extended with magic sets, say, it might lead to a very respectable goal-dependent analysis.

Codish and Demoen [10] describe a truth-table based implementation technique for *Pos* that would encode $(x_1 \leftrightarrow (x_2 \wedge x_3))$ as three tuples $\langle true, true, true\rangle$, $\langle false, \_, false\rangle$, $\langle false, false, \_\rangle$. A widening for this *Pos* analysis, $WPos$, is proposed by Codish *et al* [11] that amounts to a sub-domain of Def that cannot propagate dependencies of the form $y \leftrightarrow (y \wedge z)$, but only simple dependencies like $(x \leftrightarrow y)$. The main finding of Codish *et al* [11] is that $WPos$ looses only a small amount of precision for goal-dependent analysis of Prolog and CLP($\mathcal{R}$) programs.

9 Conclusions

We have developed the link between *Def* and *Share* to show how the *meet* of *Def* can be modelled with an efficient (quadratic) operation on *Share*. We have shown how to represent formulae succinctly with equivalence classes of sharing abstractions, and how formulae can be widened so as to avoid bad space behaviour. Putting these ideas together we have achieved a practical analysis that is fast, precise, robust and can be implemented easily in Prolog.

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. R. Bagnara. A Reactive Implementation of *Pos* using ROBDDs. In *Programming Languages: Implementation, Logics and Programs*, pages 107–121. Springer-Verlag, 1996. LNCS 1140.
3. R. Bagnara, P. Hill, and E. Zaffanella. Set-Sharing is Redundant for Pair-Sharing. In *Static Analysis Symposium*, pages 53–67. Springer-Verlag, 1997. LNCS 1302.
4. R. Bagnara and P. Schachte. Factorizing Equivalent Variable Pairs in ROBDD-Based Implementations of Pos. In *Seventh International Conference on Algebraic Methodology and Software Technology*, Amazonia, Brazil, 1999. Springer-Verlag.
5. N. Baker and H. Søndergaard. Definiteness analysis for CLP($\mathcal{R}$). In *Australian Computer Science Conference*, pages 321–332, 1993.

6. P. Bigot, S. Debray, and K. Marriott. Understanding finiteness analysis using abstract interpretation. In *Joint International Conference and Symposium on Logic Programming*, pages 735–749. MIT Press, 1992.
7. R. Bryant. Symbolic Boolean manipulation with ordered binary-decision digrams. *ACM Computing Surveys*, 24(3):293–318, 1992.
8. M. Codish. Worst Case Groundness Analysis. Technical report, Ben-Gurion University of the Negev, 1998. ftp://ftp.cs.bgu.ac.il/pub/people/mcodish/pathpos.ps.gz.
9. M. Codish. Efficient Goal Directed Bottom-up Evaluation of Logic Programs. *J. of Logic Programming (to appear)*, 1999.
10. M. Codish and B. Demoen. Analysing Logic Programs using "prop"-ositional Logic Programs and a Magic Wand. *J. Logic Programming*, 25(3):249–274, 1995.
11. M. Codish, A. Heaton, A. King, M. Abo-Zaed, and P. Hill. Widening Positive Boolean functions for Goal-dependent Groundness Analysis. Technical Report 12-98, Computing Laboratory, May 1998. http://www.cs.ukc.ac.uk/pubs/1998/589.
12. M.-M. Corsini, K. Musumbu, A. Rauzy, and B. Le Charlier. Efficient Bottom-up Abstract Interpretation of Prolog by means of Constraint Solving over Finite Domains. In *In proceedings of Programming Language Implementation and Logic Programming*, pages 75–91. Springer-Verlag, 1993. LNCS 714.
13. A. Cortesi and G. Filé. Comparison and Design of Abstract Domains for Sharing Analysis. In *Italian Conference on Logic Programming*, pages 251–265, 1993.
14. A. Cortesi, G. Filé, R. Giacobazzi, C. Palamidessi, and F. Ranzato. Complementation in Abstract Interpretation. *ACM TOPLAS*, 19(1):7–47, January 1997.
15. A. Cortesi, G. Filé, and W. Winsborough. Comparison of Abstract Interpretations. In *International Conference on Automata, Languages and Programming*, pages 521–532. Springer-Verlag, 1992. LNCS 623.
16. A. Cortesi, G. Filé, and W. Winsborough. The Quotient of an Abstract Interpretation for Comparing Static Analyses. *Theoretical Computer Science*, 202(1–2):163–192, 1998.
17. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*, pages 238–252. ACM Press, 1977.
18. P. Dart. On Derived Dependencies and Connected Databases. *J. Logic Programming*, 11(1&2):163–188, 1991.
19. S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical Program Analysis Using General Purpose Logic Programming Systems — A Case Study. In *Programming Language Design and Implementation*, pages 117–126. ACM Press, 1996.
20. C. Fecht. *Abstrakte Interpretation logischer Programme: Theorie, Implementierung, Generierung.* PhD thesis, Universität des Saarlandes, 1997. http://www.cs.uni-sb.de/RW/users/fecht/.
21. M. García de la Banda, M. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens, and W. Simoens. Global Analysis of Constraint Logic Programs. *ACM TOPLAS*, 18(5):564–614, 1996.
22. D. Jacobs and A. Langen. Static Analysis of Logic Programs. *J. Logic Programming*, 13(2 and 3):154–314, 1992.
23. N. Jones and H. Søndergaard. A semantics-based framework for the abstract interpretation of Prolog. In *Abstract Interpretation of Declarative Languages*, pages 123–142. Ellis Horwood Ltd, 1987.
24. P. Schachte. *Precise and Efficient Static Analysis of Logic Programs.* PhD thesis, Department of Computer Science, The University of Melbourne, Melbourne, Australia, 1998. (to appear).
25. H. Seidel. Personal communication on Prolog compiler integration and SML. November 1997.
26. P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Evaluation of the domain *Prop*. *J. Logic Programming*, 23(3):237–278, 1995.
27. J. Wunderwald. Memoing Evaluation by Source-to-Source Transformation. In *Logic Program Synthesis and Transformation*, pages 17–32. Springer, 1995. LNCS 1048.

Types and Subtypes for Client-Server Interactions

Simon Gay and Malcolm Hole

Department of Computer Science, Royal Holloway, Universityof London
Egham, Surrey, TW20 0EX, UK
S.Gay@dcs.rhbnc.ac.uk, M.Hole@dcs.rhbnc.ac.uk

Abstract. We define an extension of the π-calculus with a static type system which supports high-level specifications of extended patterns of communication, such as client-server protocols. Subtyping allows protocol specifications to be extended in order to describe richer behaviour; an implemented server can then be replaced by a refined implementation, without invalidating type-correctness of the overall system. We use the POP3 protocol as a concrete example of this technique.

1 Introduction

Following its early success as a framework for the investigation of the foundations of various concurrent programming styles, the π-calculus [12] has also become established as a vehicle for the exploration of type systems for concurrent programming languages [2,7,9,11,15,21]. Inter-process communication in the π-calculus is based on point-to-point transmission of messages along named channels, and many proposed type systems have started from the assignment of types to channels, so that the type of a channel determines what kind of message it can carry. Because messages can themselves be channel names, this straightforward idea leads to detailed specifications of the intended uses of channels within a system. This line of research has not been purely theoretical: the Pict programming language [16] is directly based on the π-calculus and has a rich type system which incorporates subtyping on channel types and higher-order polymorphism.

Honda et al. [5,20] have proposed π-calculus-like languages in which certain channels can be given *session types*. Such a channel, which we will call a *session channel*, is not restricted to carrying a single type of message for the whole of its lifetime; instead, its type specifies a sequence of message types. Some of the messages might indicate choices between a range of possibilities, and different choices could lead to different specifications of the types of subsequent messages; session types therefore have a branching structure. One application of session types is the specification of complex protocols, for example in client-server systems. The main contribution of the present paper is to add subtyping to a system of session types, and show that this strengthens the application to the specification of client-server protocols. The differences in syntax between

S.D. Swierstra (Ed.): ESOP/ETAPS'99, LNCS 1576, pp. 74–90, 1999.
© Springer-Verlag Berlin Heidelberg 1999

our language and the π-calculus are minimal; all the special treatment of session channels is handled by the typing rules. We anticipate that this will make it easier to achieve our next goal of incorporating our type system into a modified version of the Pict language and compiler.

Consider a server for mathematical operations, which initially offers a choice between addition and negation. A client must choose an operation and then send the appropriate number of arguments to the server, which responds by sending back the result. All communications take place on a single session channel called x, whose session type is

$$S = \&\langle \mathsf{plus} : ?[\mathsf{int}] \,.\, ?[\mathsf{int}] \,.\, ![\mathsf{int}] \,.\, \mathsf{end}, \mathsf{negate} : ?[\mathsf{int}] \,.\, ![\mathsf{int}] \,.\, \mathsf{end} \rangle.$$

More precisely, this is the type of the server side of the channel. The $\&\langle \ldots \rangle$ constructor specifies that a choice is offered between, in this case, two options, labelled plus and negate. Each label leads to a type which describes the subsequent communication on x; note that the two branches have different types, in which ?[int] indicates receiving an integer, ![int] indicates sending an integer, . is the sequencing constructor, and end indicates the end of the interaction. The client side of the channel x has a dual or complementary type, written $\overline{S}$. Explicitly,

$$\overline{S} = \oplus\langle \mathsf{plus} : ![\mathsf{int}] \,.\, ![\mathsf{int}] \,.\, ?[\mathsf{int}] \,.\, \mathsf{end}, \mathsf{negate} : ![\mathsf{int}] \,.\, ?[\mathsf{int}] \,.\, \mathsf{end} \rangle.$$

The $\oplus\langle \ldots \rangle$ constructor specifies that the client makes a choice between plus and negate. Again, each label is followed by a type which describes the subsequent interaction; the pattern of sending and receiving is the opposite of the pattern which appears on the server side.

An implementation of a maths server must use x in accordance with the type S, and an implementation of a client must use x in accordance with the type $\overline{S}$. These requirements can be enforced by static typechecking, and it is then guaranteed that no communication errors will occur at runtime. When the client chooses a label, it is guaranteed to be one of the labels offered by the server; when the client sends a subsequent message, it is guaranteed to be of the type expected by the server; and similarly when the server sends a message.

The typing rules to be introduced in Section 3 will allow the derivation of

$$x : S^1 \vdash \mathsf{server}$$

where

$$\begin{aligned} \mathsf{server} = x \triangleright \{ \; & \mathsf{plus} : x \,?\, [a : \mathsf{int}] \,.\, x \,?\, [b : \mathsf{int}] \,.\, x \,!\, [a + b] \,.\, \mathbf{0}, \\ & \mathsf{negate} : x \,?\, [a : \mathsf{int}] \,.\, x \,!\, [-a] \,.\, \mathbf{0} \}. \end{aligned}$$

The operation $\triangleright$ allows a message on x to choose between the listed alternatives. The labels are the same as those in S, and the pattern of inputs $(x \,?\, [a : \mathsf{int}] \ldots .)$ and outputs $(x \,!\, [a + b] \ldots .)$ matches that in S. The usage annotation of 1 on S indicates that only one side of x is being used by server.

One possible definition of a client is

$$\mathsf{client} = x \triangleleft \mathsf{negate} \,.\, x \,!\, [2] \,.\, x \,?\, [a : \mathsf{int}] \,.\, \mathbf{0}$$

and we can derive the typing judgement

$$x : \overline{S}^1 \vdash \mathsf{client}.$$

Note the use of $\triangleleft$ to select from the available options, and that the subsequent pattern of inputs and outputs matches the specification in $\overline{S}$. This client does not do anything with the value received from the server; more realistically, $\mathbf{0}$ would be replaced by some continuation process which used a.

The client and the server can be put in parallel using the typing rule T-Par for parallel composition:

$$\frac{x : S^1 \vdash \mathsf{server} \quad x : \overline{S}^1 \vdash \mathsf{client}}{x : S^2 \vdash \mathsf{server} \mid \mathsf{client}}$$

where the usage annotation 2 on S indicates that both sides of x are being used.

The usage annotations are necessary in order to ensure that each side of x is only used by one process. The system $\mathsf{server} \mid \mathsf{client} \mid \mathsf{client}$ is erroneous because both clients are trying to use x to communicate with the same server. If this system is executed, one client will use x to choose either plus or negate. After that, the server expects to receive an integer on x, but the other client will again use x to choose between plus and negate. This is a runtime type error of the kind that the type system is designed to avoid. We will see later that

$$\frac{x : \overline{S}^1 \vdash \mathsf{client} \quad x : \overline{S}^1 \vdash \mathsf{client}}{x : \overline{S}^1 \vdash \mathsf{client} \mid \mathsf{client}}$$

is not a valid application of the typing rule T-Par.

To avoid runtime type errors we must ensure that session channels are used linearly [3]. Our typing rules use techniques similar to those of Kobayashi et al. [9] to enforce linearity. The type system also allows non-session types to be specified, and there are no restrictions on how many processes may use them. For example, $y : \hat{}[\mathsf{int}]$ is a channel which can be used freely to send or receive integers.

How, then, can we implement a server which can be used by more than one client? The solution is for each client to create a session channel which it will use for its own interaction with the server. The server consists of a replicated thread process; each thread receives a session channel along a channel called port and uses it to interact with a client.

$$\begin{aligned}
\mathsf{thread} &= \mathsf{port}\,?\,[x : S^1]\,.\,\mathsf{server} \\
\mathsf{newserver} &= !\mathsf{thread} \\
\mathsf{client1body} &= y \triangleleft \mathsf{negate} : y\,!\,[2]\,.\,y\,?\,[a : \mathsf{int}]\,.\,\mathbf{0} \\
\mathsf{client1} &= (\nu y : S^2)\mathsf{port}\,!\,[y]\,.\,\mathsf{client1body} \\
\mathsf{client2body} &= z \triangleleft \mathsf{plus} : z\,!\,[1]\,.\,z\,!\,[2]\,.\,z\,?\,[b : \mathsf{int}]\,.\,\mathbf{0} \\
\mathsf{client2} &= (\nu z : S^2)\mathsf{port}\,!\,[z]\,.\,\mathsf{client2body}
\end{aligned}$$

Now

$$\mathsf{port} : \hat{}[S^1] \vdash \mathsf{newserver} \mid \mathsf{client1} \mid \mathsf{client2}$$

is a valid typing judgement. Because port does not have a session type, it can be used by all three processes. When this system is executed, client1 sends the local channel y to one copy of thread; the standard π-calculus scope extrusion allows the scope of y to expand to include that copy of thread, and then client1body and server have a private interaction on y. Similarly client2 and another copy of thread use the private channel z for their communication. Notice that y is a session channel with usage 2 in client1, and indeed both sides of y are used: the side whose type is S is used by being sent away on port, and the side whose type is $\overline{S}$ is used for communication by client1body.

The final ingredient of our type system is subtyping. On non-session types, subtyping is defined exactly as in Pierce and Sangiorgi's type system for the π-calculus [15]: if $\forall i \in \{1,\ldots,n\}.T_i \leqslant U_i$ then $\hat{}[T_1,\ldots,T_n] \leqslant ?[U_1,\ldots,U_n]$ and $\hat{}[U_1,\ldots,U_n] \leqslant ![T_1,\ldots,T_n]$. Channels whose type permits both input and output ($\hat{}$) can be used in positions where just input or just output is required. We also have $?[T_1,\ldots,T_n] \leqslant ?[U_1,\ldots,U_n]$ and $![U_1,\ldots,U_n] \leqslant ![T_1,\ldots,T_n]$; recall that input behaves covariantly and output behaves contravariantly.

Subtyping on sequences $T_1.\cdots.T_n$ is defined pointwise, again with ? acting covariantly and ! acting contravariantly. More interesting is the definition of subtyping for branch and choice types. If a process needs a channel of type $\&\langle l_1:T_1,\ldots,l_n:T_n\rangle$ which allows it to offer a choice from $\{l_1,\ldots,l_n\}$, then it can safely use a channel of type $\&\langle l_1:T_1,\ldots,l_m:T_m\rangle$, where $m \leqslant n$, instead. The channel type prevents the process from ever receiving labels $l_{m+1},\ldots,l_n$ but every label that can be received will be understood. Furthermore, a channel of type $\&\langle l_1:S_1,\ldots,l_m:S_m\rangle$ can be used if each $S_i \leqslant T_i$, as this means that after the choice has been made the continuation process uses a channel of type S_i instead of T_i and this is safe. In Section 5 we will see how subtyping can be used to decribe modifications to the specification of a server.

The remainder of the paper is organised as follows. Section 2 defines the syntax of processes and types, and some basic operations on type environments. The typing rules are presented in Section 3. Section 4 defines the operational semantics of the language and states the main technical results leading to type soundness. Section 5 uses our type system to specify the POP3 protocol, and discusses the role of subtyping. Finally we discuss related work, and outline our future plans, in Section 6.

2 Syntax and Notation

Our language is based on a polyadic π-calculus with output prefixing [12]. We omit the original π-calculus choice construct $P+Q$, partly in order to keep the language close to the core of Pict [16]. However, we have the constructs introduced in Section 1 for choosing between a collection of labelled processes, as proposed by Honda et al. [5,20]. We also omit the matching construct, which allows channel names to be tested for equality, again because it is not present in core Pict. The inclusion of output prefixing is different from many recent presentations of the π-calculus, but it is essential because our type system must

be able to impose an order on separate outputs on the same channel. It is convenient to add a conditional process expression, written $\mathsf{if}\, b \,\mathsf{then}\, P \,\mathsf{else}\, Q$ where b is a boolean value, and therefore we also have a ground type of booleans; other ground types, such as int as used in the examples in Section 1, could be added along with appropriate primitive operations. As is standard, we use the replication operator ! instead of recursive process definitions.

The type system has separate constructors for input-only, output-only and dual-capability channels, as suggested by Pierce and Sangiorgi [15]. It also has constructors for session types, as proposed by Honda et al. [5,20]. The need for linear control of session channels leads to the usage annotations on session types, which play a similar role to the polarities of Kobayashi et al. [9]. Subtyping will be defined in Section 3.

In general we use lower case letters for channel names, $l_1, \ldots, l_n$ for labels of choices, upper case P, Q, R for processes, and upper case T, U etc. for types. We write $\tilde{x}$ for a finite sequence $x_1, \ldots, x_n$ of names, and $\tilde{x} : \tilde{T}$ for a finite sequence $x_1 : T_1, \ldots, x_n : T_n$ of typed names.

2.1 Processes

The syntax of processes is defined by the following grammar. Note that T and $\tilde{T}$ stand for types and lists of types, which have not yet been defined.

$$
\begin{array}{lll}
P ::= \mathbf{0} & & \\
\quad \mid\ P \mid Q & & \mid\ x \triangleright \{l_1 : P_1, \ldots, l_n : P_n\} \\
\quad \mid\ x\,?\,[\tilde{y} : \tilde{T}]\,.\,P & & \mid\ x \triangleleft l\ .\ P \\
\quad \mid\ x\,!\,[\tilde{y}]\,.\,P & & \mid\ !P \\
\quad \mid\ (\nu x : T)P & & \mid\ \mathsf{if}\, x \,\mathsf{then}\, P \,\mathsf{else}\, Q
\end{array}
$$

Most of this syntax is fairly standard. $\mathbf{0}$ is the inactive process, $\mid$ is parallel composition, $(\nu x : T)P$ declares a local name x of type T for use in P, and $!P$ represents a potentially infinite supply of copies of P. $x\,?\,[\tilde{y} : \tilde{T}]\,.\,P$ receives the names $\tilde{y}$, which have types $\tilde{T}$, along the channel x, and then executes P. $x\,!\,[\tilde{y}]\,.\,P$ outputs the names $\tilde{y}$ along the channel x and then executes P. There should be no confusion between the use of ! for output and its use for replication, as the surrounding syntax is quite different in each case. $x \triangleright \{l_1 : P_1, \ldots, l_n : P_n\}$ offers a choice of subsequent behaviours—one of the P_i can be selected as the continuation process by sending the appropriate label l_i along the channel x, as explained in Section 1. $x \triangleleft l\ .\ P$ sends the label l along x in order to make a selection from an offered choice, and then executes P. The conditional expression has already been mentioned.

We define free and bound names as usual: x is bound in $(\nu x : T)P$, the names in $\tilde{y}$ are bound in $x\,?\,[\tilde{y} : \tilde{T}]\,.\,P$, and all other occurrences are free. We then define α-equivalence as usual, and identify processes which are α-equivalent. We also define an operation of substitution of names for names: $P\{\tilde{x}/\tilde{y}\}$ denotes P with the names $x_1, \ldots, x_n$ simultaneously substituted for $y_1, \ldots, y_n$, assuming that bound names are renamed if necessary to avoid capture of substituting names.

As usual we define a *structural congruence* relation, written $\equiv$, which helps to define the operational semantics. It is the smallest congruence (on α-equivalence classes of processes) closed under the following rules.

$$
\begin{aligned}
P \mid \mathbf{0} &\equiv P && \text{S-Unit}\\
P \mid Q &\equiv Q \mid P && \text{S-Comm}\\
P \mid (Q \mid R) &\equiv (P \mid Q) \mid R && \text{S-Assoc}\\
(\nu x : T)P \mid Q &\equiv (\nu x : T)(P \mid Q) \text{ if } x \text{ is not free in } Q && \text{S-Extr}\\
(\nu x : T)\mathbf{0} &\equiv \mathbf{0} && \text{S-Nil}\\
(\nu x : T)(\nu y : U)P &\equiv (\nu y : U)(\nu x : T)P && \text{S-Perm}\\
!P &\equiv P|!P && \text{S-Rep}\\
x \triangleright \{l_1 : P_1, \ldots, l_n : P_n\} &\equiv x \triangleright \{l_{\sigma(1)} : P_{\sigma(1)}, \ldots, l_{\sigma(n)} : P_{\sigma(n)}\} && \text{S-Offer}
\end{aligned}
$$

In rule S-Offer, σ is a permutation on $\{1, \ldots, n\}$.

2.2 Types

The syntax of types is defined by the following grammar.

$$
\begin{array}{lll}
\text{Ground types} & G & ::= \mathsf{bool}\\
\text{Channel types} & C & ::= ?[T_1, \ldots, T_n]\\
 & & \mid\ ![T_1, \ldots, T_n]\\
 & & \mid\ \hat{}[T_1, \ldots, T_n]\\
\text{Session types} & S & ::= \mathsf{end}\\
 & & \mid\ ?[T_1, \ldots, T_n] \,.\, S\\
 & & \mid\ ![T_1, \ldots, T_n] \,.\, S\\
 & & \mid\ \&\langle l_1 : S_1, \ldots, l_n : S_n\rangle\\
 & & \mid\ \oplus\langle l_1 : S_1, \ldots, l_n : S_n\rangle\\
\text{Types} & T & ::= G|C|A\\
 & & \mid\ X (\text{type variable})\\
 & & \mid\ \mu X.T (\text{recursive type})\\
\text{Annotated session types} & A & ::= S^1|S^2
\end{array}
$$

The *usage annotation*, or just *usage*, of a session type indicates how a channel of that type can be used: if $x : S^1$ then x can only be used as specified by S, but if $x : S^2$ then both sides of x can be used, including the side described by $\overline{S}$. We omit usage annotations from end, and often omit usage annotations of 1.

We define the *unwinding* of a recursive type: $\mathrm{unwind}(\mu X.T) = T\{\mu X.T/X\}$.

If T is a type then $\overline{T}$ is the dual (or complementary) type of T, defined inductively as follows.

$$
\begin{aligned}
\overline{\&\langle l_1 : S_1, \ldots, l_n : S_n\rangle} &= \oplus\langle l_1 : \overline{S_1}, \ldots, l_n : \overline{S_n}\rangle\\
\overline{\oplus\langle l_1 : S_1, \ldots, l_n : S_n\rangle} &= \&\langle l_1 : \overline{S_1}, \ldots, l_n : \overline{S_n}\rangle\\
\overline{\mathsf{bool}} = \mathsf{bool} \qquad & \qquad \overline{?[\tilde{T}]} = ![\tilde{T}]\\
\overline{\mathsf{end}} = \mathsf{end} \qquad & \qquad \overline{![\tilde{T}]} = ?[\tilde{T}]\\
\overline{X} = X \qquad & \qquad \overline{\mu X.T} = \mu X.\overline{T}
\end{aligned}
$$

S^2 and $\overline{S}^2$ both describe a session channel of which both ends are being used. We adopt the convention that when S^2 is written, S is either a branching type or begins with an input. We say that a session type is *complete* if it has usage 2; it is *incomplete* if it is end or has usage 1.

2.3 Environments

An environment is a set of typed names, written $x_1 : T_1, \ldots, x_n : T_n$. We use Γ, Δ etc. to stand for environments. We assume that all the names in an environment are distinct. We write $x \in \Gamma$ to indicate that x is one of the names appearing in Γ, and then write $\Gamma(x)$ for the type of x in Γ. When $x \notin \Gamma$ we write $\Gamma, x : T$ for the environment formed by adding $x : T$ to the set of typed names in Γ. When Γ and Δ have disjoint sets of names, we write Γ, Δ for their union. Implicitly, *true* : bool and *false* : bool appear in every environment.

The partial operation + on types is defined by

$$\begin{aligned} T + T &= T \quad \text{if } T \text{ is a ground type, a channel type, or } \mathsf{end} \\ S^1 + \overline{S}^1 &= S^2 \quad \text{if } S \text{ is a session type} \end{aligned}$$

and is undefined in all other cases.

The partial operation +, combining a typed name with an environment, is defined as follows:

$$\begin{aligned} \Gamma + x : T &= \Gamma, x : T && \text{if } x \notin \Gamma \\ (\Gamma, x : T) + x : U &= \Gamma, x : (T + U) && \text{if } T + U \text{ is defined} \end{aligned}$$

and is undefined in all other cases.

We extend + to a partial operation on environments by defining

$$\Gamma + (x_1 : T_1, \ldots, x_n : T_n) = (\cdots(\Gamma + x_1 : T_1) + \cdots +) + x_n : T_n$$

We say that an environment is *unlimited* if it contains no session types except for end.

3 The Type System

3.1 Subtyping

The principles behind the definition of subtyping have been described in Section 1. Figure 1 defines the subtype relation formally by means of a collection of inference rules for judgements of the form $\Sigma \vdash T \leqslant U$, where Σ ranges over finite sets of instances of $\leqslant$. When $\emptyset \vdash T \leqslant U$ is derivable we simply write $T \leqslant U$. The inference rules can be interpreted as an algorithm for checking whether $T \leqslant U$ for given T and U, as follows. Beginning with the goal $\emptyset \vdash T \leqslant U$, apply the rules upwards to generate subgoals; pattern matching on the structure of T and U determines which rule to use, except that the rule AS-Assump should always be used, causing the current subgoal to succeed, if it is applicable. If

both AS-Rec-L and AS-Rec-R are applicable then they should be used in either order. If a subgoal is generated which does not match any of the rules, the algorithm returns false.

Pierce and Sangiorgi [15] give two definitions of their subtype relation: one by means of inference rules (as in Figure 1) and one as a form of type simulation, defined coinductively. The subtyping algorithm derived from the inference rules can then be proved sound and complete with respect to the coinductive definition, while the coinductive definition permits straightforward proofs of transitivity and reflexivity of the subtype relation. In the same way, we can characterize our subtype relation coinductively, prove soundness and completeness of the subtyping algorithm, and prove transitivity and reflexivity; due to space constraints, we have omitted the details from the present paper.

The subtype relation is defined on non-annotated types, but annotations preserve subtyping: if $i \in \{1, 2\}$ then $S^i \leqslant T^i$ if and only if $S \leqslant T$.

If $\tilde{T}$ and $\tilde{U}$ have the same length, n, and $\forall i \in \{1, \ldots, n\}.T_i \leqslant U_i$, we write $\tilde{T} \leqslant \tilde{U}$.

3.2 Typing rules

The typing rules are defined in Figure 2. Note that a judgement of the form $\Gamma \vdash x : T \leqslant U$ means $x : T \in \Gamma$ and $T \leqslant U$. Subtyping appears in the hypotheses of rules T-Out, T-OutSeq, T-In and T-InSeq, where it must be possible to promote the type of the channel to the desired input or output type. It appears less explicitly in rules T-Offer and T-Choose, where the type of x must include enough labels for the choice being constructed.

Each typing rule is only applicable when any instances of $+$ which it contains are actually defined. This ensures that the environment correctly records the use being made of session channels. Consider again the two applications of T-Par from Section 1.

$$\frac{x : S^1 \vdash \mathsf{server} \quad x : \overline{S}^1 \vdash \mathsf{client}}{x : S^2 \vdash \mathsf{server} \mid \mathsf{client}} \qquad \frac{x : \overline{S}^1 \vdash \mathsf{client} \quad x : \overline{S}^1 \vdash \mathsf{client}}{x : \overline{S}^1 \vdash \mathsf{client} \mid \mathsf{client}}$$

The first is correct because $S^1 + \overline{S}^1 = S^2$. The second is incorrect because $\overline{S}^1 + \overline{S}^1$ is not defined; this prevents the session channel x from being used simultaneously by both copies of client.

Notice also that in rules T-Out and T-OutSeq, the names being output are added to the environment; this means that if a session channel is output then the part of its usage which is given away cannot be used again by the remainder of the process. This allows a process to begin a communication on a session channel, then delegate the rest of the session to another process by sending it the channel; of course, the first process must not use the channel again. Such behaviour arises when a recursive process, which uses a session channel (of a recursive type), is represented in terms of replication: when the next instance of the recursive process is invoked, the session channel must be passed on.

$$\frac{T \leqslant U \in \Sigma}{\Sigma \vdash T \leqslant U}\ \text{AS-Assump}$$

$$\frac{}{\Sigma \vdash \mathsf{bool} \leqslant \mathsf{bool}}\ \text{AS-Bool} \qquad \frac{}{\Sigma \vdash \mathsf{end} \leqslant \mathsf{end}}\ \text{AS-End}$$

$$\frac{\Sigma \vdash \tilde{T} \leqslant \tilde{U}}{\Sigma \vdash ?[\tilde{T}] \leqslant ?[\tilde{U}] \quad \Sigma \vdash \hat{}[\tilde{T}] \leqslant ?[\tilde{U}]}\ \text{AS-In}$$

$$\frac{\Sigma \vdash \tilde{U} \leqslant \tilde{T}}{\Sigma \vdash ![\tilde{T}] \leqslant ![\tilde{U}] \quad \Sigma \vdash \hat{}[\tilde{T}] \leqslant ![\tilde{U}]}\ \text{AS-Out}$$

$$\frac{\Sigma \vdash \tilde{T} \leqslant \tilde{U} \text{ and } \Sigma \vdash \tilde{U} \leqslant \tilde{T}}{\Sigma \vdash \hat{}[\tilde{T}] \leqslant \hat{}[\tilde{U}]}\ \text{AS-InOut}$$

$$\frac{\Sigma \vdash V \leqslant W \quad \Sigma \vdash \tilde{T} \leqslant \tilde{U}}{\Sigma \vdash ?[\tilde{T}] \,.\, V \leqslant ?[\tilde{U}] \,.\, W}\ \text{AS-InSeq}$$

$$\frac{\Sigma \vdash V \leqslant W \quad \Sigma \vdash \tilde{U} \leqslant \tilde{T}}{\Sigma \vdash ![\tilde{T}] \,.\, V \leqslant ![\tilde{U}] \,.\, W}\ \text{AS-OutSeq}$$

$$\frac{m \leqslant n \quad \forall i \in \{1, \ldots, m\}. \Sigma \vdash S_i \leqslant T_i}{\Sigma \vdash \&\langle l_1 : S_1, \ldots, l_m : S_m \rangle \leqslant \&\langle l_1 : T_1, \ldots, l_n : T_n \rangle}\ \text{AS-Branch}$$

$$\frac{m \leqslant n \quad \forall i \in \{1, \ldots, m\}. \Sigma \vdash T_i \leqslant S_i}{\Sigma \vdash \oplus\langle l_1 : S_1, \ldots, l_n : S_n \rangle \leqslant \oplus\langle l_1 : T_1, \ldots, l_m : T_m \rangle}\ \text{AS-Choice}$$

$$\frac{\Sigma, \mu X.S \leqslant T \vdash \text{unwind}(\mu X.S) \leqslant T}{\Sigma \vdash \mu X.S \leqslant T}\ \text{AS-Rec-L}$$

$$\frac{\Sigma, T \leqslant \mu X.S \vdash T \leqslant \text{unwind}(\mu X.S)}{\Sigma \vdash T \leqslant \mu X.S}\ \text{AS-Rec-R}$$

Fig. 1. Inference rules for subtyping

$$\frac{\Gamma \text{ unlimited}}{\Gamma \vdash \mathbf{0}}\ \text{T-Nil} \qquad \frac{\Gamma \vdash P \quad \Delta \vdash Q}{\Gamma + \Delta \vdash P \mid Q}\ \text{T-Par}$$

$$\frac{\Gamma, x : T \vdash P \quad \text{if } T \text{ is a session type it must be complete}}{\Gamma \vdash (\nu x : T)P}\ \text{T-New}$$

$$\frac{\Gamma \vdash P \quad \Gamma \vdash x \leqslant ![\tilde{T}]}{\Gamma + \tilde{y} : \tilde{T} \vdash x\,![\tilde{y}]\,.\,P}\ \text{T-Out} \qquad \frac{\Gamma, \tilde{y} : \tilde{T} \vdash P \quad \Gamma \vdash x \leqslant ?[\tilde{T}]}{\Gamma \vdash x\,?\,[\tilde{y} : \tilde{T}]\,.\,P}\ \text{T-In}$$

$$\frac{\Gamma, x : S \vdash P \quad S \text{ is incomplete} \quad \tilde{U} \leqslant \tilde{T}}{(\Gamma, x : ![\tilde{T}]\,.\,S) + \tilde{y} : \tilde{U} \vdash x\,![\tilde{y}]\,.\,P}\ \text{T-OutSeq}$$

$$\frac{\Gamma, x : S, \tilde{y} : \tilde{U} \vdash P \quad S \text{ is incomplete} \quad \tilde{T} \leqslant \tilde{U}}{\Gamma, x : ?[\tilde{T}]\,.\,S \vdash x\,?\,[\tilde{y} : \tilde{U}]\,.\,P}\ \text{T-InSeq}$$

$$\frac{\Gamma, x : S_1 \vdash P_1 \ldots \Gamma, x : S_n \vdash P_n \quad \text{each } S_i \text{ is incomplete} \quad m \leqslant n}{\Gamma, x : \&\langle l_1 : S_1, \ldots, l_m : S_m \rangle^1 \vdash x \triangleright \{l_1 : P_1, \ldots, l_n : P_n\}}\ \text{T-Offer}$$

$$\frac{\Gamma, x : {S_i}^{t_i} \vdash P \quad S_i = \mathsf{end} \text{ or } t_i = 1}{\Gamma, x : \oplus\langle l_1 : S_1, \ldots, l_n : S_n \rangle^1 \vdash x \triangleleft l_i\ .\ P}\ \text{T-Choose}$$

$$\frac{\Gamma \vdash x : \mathsf{bool} \quad \Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash \ \mathsf{if}\ x\ \mathsf{then}\ P\ \mathsf{else}\ Q}\ \text{T-Cond} \qquad \frac{\Gamma \vdash P \quad \Gamma \text{ unlimited}}{\Gamma \vdash !P}\ \text{T-Rep}$$

Fig. 2. Typing rules

$$\frac{}{x\,?\,[\tilde{y} : \tilde{T}]\,.\,P \mid x\,![\tilde{z}]\,.\,Q \longrightarrow P\{\tilde{z}/\tilde{y}\} \mid Q}\ \text{R-Comm}$$

$$\frac{i \in \{1, \ldots, n\}}{x \triangleright \{l_1 : P_1, \ldots, l_n : P_n\} \mid x \triangleleft l_i\ .\ Q \longrightarrow P_i \mid Q}\ \text{R-Select}$$

$$\frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q}\ \text{R-Par} \qquad \frac{P' \equiv P \quad P \longrightarrow Q \quad Q \equiv Q'}{P' \longrightarrow Q'}\ \text{R-Cong}$$

$$\frac{P \longrightarrow P'}{(\nu x : T)P \longrightarrow (\nu x : T)P'}\ \text{R-New}$$

$$\frac{}{\mathsf{if}\ \mathit{true}\ \mathsf{then}\ P\ \mathsf{else}\ Q \longrightarrow P}\ \text{R-True} \qquad \frac{}{\mathsf{if}\ \mathit{false}\ \mathsf{then}\ P\ \mathsf{else}\ Q \longrightarrow Q}\ \text{R-False}$$

Fig. 3. The reduction relation

Finally, note that a session type $T.\mathsf{end}$ effectively specifies a linear non-session channel of type T as used in [9].

4 Operational Semantics

As usual for π-calculus-like languages, the operational semantics is defined by means of a reduction relation [11]. $P \longrightarrow Q$ means that the process P reduces to the process Q by executing a single communication step (or evaluating a conditional expression). The reduction relation is the smallest relation closed under the rules in Figure 3, most of which are standard. The rules R-Comm and R-Select introduce communication steps; R-Comm is standard and R-Select introduces communications which select labelled options. Note that R-Comm applies to both session and non-session channels, as there is no indication of the type of x in either process.

The usual way of proving type soundness is first to prove a subject reduction theorem: if $\Gamma \vdash P$ and $P \longrightarrow Q$ then there is an environment Δ such that $\Delta \vdash Q$. Then, one proves that if $\Gamma \vdash P$ then the immediately available communications in P do not cause type errors. Together these results imply that a well-typed process can be executed safely through any sequence of reduction steps. However, the presence of subtyping means that examining Γ is not sufficient to determine what constitutes correct use of names in P; different occurrences of a single name in P might be constrained to have different types. For example, the typed process

$$a : \hat{}[![\mathsf{bool}]], b : \hat{}[?[\mathsf{bool}]], x : \hat{}[\mathsf{bool}] \vdash \\ a\,![x]\,.\,b\,![x]\,.\,\mathbf{0} \mid a\,?\,[y : ![\mathsf{bool}]]\,.\,P \mid b\,?\,[z : ?[\mathsf{bool}]]\,.\,Q$$

reduces in two steps to, essentially, $x : \hat{}[\mathsf{bool}] \vdash P\{x/y\} \mid Q\{x/z\}$, and occurrences of x in P have type $![\mathsf{bool}]$ but those in Q have type $?[\mathsf{bool}]$.

To address this difficulty, we adopt Pierce and Sangiorgi's technique of introducing *tagged* processes [15], written E, F, etc. instead of P, Q, etc. The syntax of tagged processes is identical to that of ordinary processes except that *all* occurrences of names are typed, for example $(x : T)\,![\tilde{x} : \tilde{U}]\,.\,E$. Structural congruence is defined on tagged processes by the same rules, with tags added, as for untagged processes. We also introduce tagged typing rules, defining judgements of the form $\Gamma \vdash E$. The tagged typing rules are essentially the same as the untagged typing rules; a typical example is the rule TT-Out.

$$\frac{\Gamma \vdash E \quad \Gamma \vdash x \leqslant T \leqslant ![\tilde{U}] \quad \tilde{V} \leqslant \tilde{U}}{\Gamma + \tilde{y} : \tilde{V} \vdash (x : T)\,![\tilde{y} : \tilde{U}]\,.\,E}\ \text{TT-Out}$$

Note that the type declared for a name in the environment must be a subtype of the type with which that name is tagged in the process.

The tagged reduction relation is written $\Gamma \vdash E \longrightarrow \Delta \vdash F$ and is defined by the rules in Figure 4, together with tagged versions of the rules R-Par, R-Cong, R-New, R-True and R-False.

$$\frac{T \leqslant ?[\tilde{U}] \quad V \leqslant ![\tilde{U}] \quad \tilde{W} \leqslant \tilde{U}}{\begin{array}{c}\Gamma, x : \hat{}[\tilde{S}] \vdash (x : T)\,?\,[\tilde{y} : \tilde{U}]\,.\,P \mid (x : V)\,!\,[\tilde{z} : \tilde{W}]\,.\,Q \\ \longrightarrow \Gamma, x : \hat{}[\tilde{S}] \vdash P\{\tilde{z}/\tilde{y}\} \mid Q\end{array}}\ \text{TR-Comm}$$

$$\frac{S \leqslant ?[\tilde{U}]\,.\,R \quad V \leqslant ![\tilde{Q}]\,.\,\overline{R} \quad \tilde{W} \leqslant \tilde{U}}{\begin{array}{c}\Gamma, x : ?[\tilde{T}]\,.\,R^2 \vdash (x : S)\,?\,[\tilde{y} : \tilde{U}]\,.\,P \mid (x : V)\,!\,[\tilde{z} : \tilde{W}]\,.\,Q \\ \longrightarrow \Gamma, x : R^2 \vdash P\{\tilde{z}/\tilde{y}\} \mid Q\end{array}}\ \text{TR-CommSeq}$$

$$\frac{m \leqslant n \quad l \in \{l_1, \ldots, l_n\}}{\begin{array}{c}\Gamma, x : \&\langle l_1 : T_1, \ldots, l_n : T_n\rangle^2 \vdash (x : S) \triangleright \{l_1 : P_1, \ldots, l_n : P_n\} \mid (x : V) \triangleleft l\,.\,Q \\ \longrightarrow \Gamma, x : T_i \vdash P_i \mid Q\end{array}}\ \text{TR-Select}$$

Fig. 4. The tagged reduction relation (selected rules)

The function Erase from tagged processes to untagged processes simply removes the extra type information. The definition is straightforward, for example

$$\mathrm{Erase}((x : T)\,!\,[\tilde{y} : \tilde{U}]\,.\,E) = x\,!\,[\tilde{y}]\,.\,\mathrm{Erase}(E).$$

Theorem 1 (Tagged Subject Reduction). *If $\Gamma \vdash E \longrightarrow \Delta \vdash F$ and $\Gamma \vdash E$ is derivable then $\Delta \vdash F$ is derivable.*

Proof. By induction on the derivation of $\Gamma \vdash E \longrightarrow \Delta \vdash F$. The assumption that $\Gamma \vdash E$ is derivable provides the information about the components of F which is needed to build a derivation of $\Delta \vdash F$.

Observe that the Tagged Subject Reduction Theorem guarantees that the tagged reduction relation is well-defined as a relation on *derivable* tgged typing judgements.

Lemma 1. *If $\Gamma \vdash P$ is a derivable untagged typing judgement, then there is a tagged process E such that $P = \mathrm{Erase}(E)$ and $\Gamma \vdash E$ is a derivable tagged typing judgement.*

Proof. We can define a function $\mathrm{Tag}_\Gamma(P)$, by induction on the structure of P, which essentially tags every name in P with its exact type as declared in Γ or by a binding ν or input. The presence of session types causes a slight complication: if $x : S^2 \in \Gamma$ and x is used in both P and Q, then $\mathrm{Tag}_\Gamma(P \mid Q)$ must ensure that x is tagged with S^1 in P and with $\overline{S}^1$ in Q, or vice versa. Essentially the same problem is encountered in linear type inference [10], and we use the same solution: $\mathrm{Tag}_\Gamma(P)$ returns a pair (P', Γ') where P' is a tagged process and Γ' differs from Γ only by the possible removal of some usages of session types. Then

$\mathrm{Tag}_\Gamma(P \mid Q) = (P' \mid Q', \Gamma'')$ where $\mathrm{Tag}_\Gamma(P) = (P'.\Gamma')$ and $\mathrm{Tag}_{\Gamma'}(Q) = (Q', \Gamma'')$. When $\Gamma \vdash P$ and $\mathrm{Tag}_\Gamma(P) = (P', \Gamma')$ we have that Γ' is unlimited (so all session types have been removed), $\mathrm{ok}_\Gamma(P')$ and $P = \mathrm{Erase}(P')$.

Theorem 2. *If $\Gamma \vdash P$ is derivable and $\Gamma \vdash E$ is derivable and $P = \mathrm{Erase}(E)$ and $P \longrightarrow^* Q$ then there exists $\Delta \vdash F$ such that $\Gamma \vdash E \longrightarrow^* \Delta \vdash F$ and $Q = \mathrm{Erase}(F)$.*

Proof. By breaking the sequence of reductions into individual steps, and showing that the result holds for each step; the latter fact can be proved by induction on the derivation of the reduction step.

The Tagged Subject Reduction Theorem, Lemma 1 and Theorem 2 imply that any sequence of reductions from a well-typed untagged process can be mirrored by a sequence of reductions from a well-typed tagged process. The final theorem establishes that well-typed tagged processes do not contain any immediate possibilities for incorrect communication. It follows easily from the taggd typing rules; most of the work in proving type soundness is concentrated into the proof of the Tagged Subject Reduction Theorem. Each case of the conclusion shows that whenever a tagged process appears to contain a potential reduction, the preconditions for the relevant tagged reduction rule are satisfied and the reduction can safely be carried out.

Theorem 3. *If P is a tagged process, $\Gamma \vdash \mathrm{Erase}(P)$ and $\mathrm{ok}_\Gamma(P)$, then*

1. *if $P \equiv (\nu\tilde{x} : \tilde{X})((a : T)\,?\,[\tilde{y} : \tilde{U}]\,.\,P_1 \mid (a : V)\,!\,[\tilde{z} : \tilde{W}]\,.\,P_2 \mid Q)$ and T is not a session type then the declaration $a : S$ occurs in either Γ or $\tilde{x} : \tilde{X}$, with $S \leqslant T \leqslant ?[\tilde{U}]$ and $S \leqslant V \leqslant ![\tilde{W}]$ and $\tilde{W} \leqslant \tilde{U}$.*
2. *if $P \equiv (\nu\tilde{x} : \tilde{X})((a : T\,.\,T')\,?\,[\tilde{y} : \tilde{U}]\,.\,P_1 \mid (a : V\,.\,V')\,!\,[\tilde{z} : \tilde{W}]\,.\,P_2 \mid Q)$ then the declaration $a : S^2$ occurs in either Γ or $\tilde{x} : \tilde{X}$, with $\overline{S} \leqslant T\,.\,T'$ and $S \leqslant V\,.\,V'$ and $T \leqslant ![\tilde{U}]$ and $V \leqslant ?[\tilde{W}]$ and $\tilde{U} \leqslant \tilde{W}$.*
3. *if $P \equiv (\nu\tilde{x} : \tilde{X})((a : T) \triangleright \{l_1 : P_1, \ldots, l_m : P_m\} \mid (a : V) \triangleleft l\ .\ P_0 \mid Q)$ then the declaration $a : S$ occurs in either Γ or $\tilde{x} : \tilde{X}$, with $S \leqslant T$ and $T = \&\langle l_1 : T_1, \ldots, l_n : T_n\rangle$ and $n \leqslant m$ and $\overline{S} \leqslant V$ and $V = \oplus\langle l_1 : V_1, \ldots, l_r : V_r\rangle$ and $r \leqslant n$ and $l \in \{l_1, \ldots, l_r\}$.*
4. *if $P \equiv (\nu\tilde{x} : \tilde{X})(\,\mathsf{if}\ a\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \mid Q)$ then the declaration $a : \mathsf{bool}$ occurs in either Γ or $\tilde{x} : \tilde{X}$.*

If we take a well-typed untagged process and convert it into a tagged process, no reduction sequence can lead to a type error. Because every reduction sequence from a well-typed untagged process can be matched by a reduction sequence from a well-typed tagged process, we conclude that no type errors can result from executing a well-typed untagged process.

5 The POP3 Protocol

As a more substantial example, we will now use our type system to specify the POP3 protocol [13]. This protocol is typically used by electronic mail software

$$
\begin{aligned}
A = \mu X.\&\langle\, & \mathsf{quit}:\oplus\langle \mathsf{ok}:![\mathsf{str}]\,.\,\mathsf{end}\rangle,\\
& \mathsf{user}:?[\mathsf{str}]\,.\,\oplus\langle\, \mathsf{error}:![\mathsf{str}]\,.\,X,\\
& \qquad \mathsf{ok}:![\mathsf{str}]\,.\,\&\langle\, \mathsf{quit}:\oplus\langle \mathsf{ok}:![\mathsf{str}]\,.\,\mathsf{end}\rangle,\\
& \qquad\qquad \mathsf{pass}:?[\mathsf{str}]\,.\,\oplus\langle\, \mathsf{error}:![\mathsf{str}]\,.\,X,\\
& \qquad\qquad\qquad \mathsf{ok}:![\mathsf{str}]\,.\,T\rangle\rangle\rangle\\
T = \mu X.\&\langle\, & \mathsf{stat}:\oplus\langle \mathsf{ok}:![\mathsf{int},\mathsf{int}]\,.\,X\rangle,\\
& \mathsf{retr}:?[\mathsf{int}]\,.\,\oplus\langle\, \mathsf{ok}:![\mathsf{str}]\,.\,![\mathsf{str}]\,.\,X,\\
& \qquad \mathsf{error}:![\mathsf{str}]\,.\,X\rangle,\\
& \mathsf{quit}:\oplus\langle \mathsf{ok}:![\mathsf{str}]\,.\,\mathsf{end}\rangle\rangle
\end{aligned}
$$

Fig. 5. Types for the POP3 protocol

to download new messages from a remote mailbox, so that they can be read and processed locally; it does not deal with sending messages or routing messages through a network. A POP3 server requires a client to authenticate itself by means of the user and pass commands. A client may then use commands such as stat to obtain the status of the mailbox, retr to retrieve a particular message, and quit to terminate the session. Some of these commands require additional information to be sent, for example the number of a message. We have omitted several POP3 commands from our description, but it is straightforward to fill in the missing ones.

To specify the behaviour of a channel which can be used for a POP3 session, we use the type definitions in Figure 5: A describes interactions with the authentication state, and T describes interactions with the transaction state. These definitions are for the server side of the channel, and we assume that there is a ground type str of strings. These definitions illustrate the complex structure possible for session types, and show the use of recursive types to describe repetitive session behaviour. The server both offers and makes choices, in contrast to the example in Section 1. After receiving a command (a label) from the client, the server can respond with either ok or error (except for the quit command, which always succeeds and does not allow an error response). The client implements an interaction of type $\overline{A}$, and therefore must offer a choice between ok and error when waiting for a response to a command. In the published description of the protocol, ok and error responses are simply strings prefixed with +OK or -ERR. This does not allow us to replace the corresponding $\oplus$ by $![\mathsf{str}]$ in the above definitions because the different continuation types after ok and error are essential for an accurate description of the protocol's behaviour. We specify a string message as well, because a POP3 server is allowed to provide extra information such as an error code.

As in Section 1 we could implement a process POP3body such that $x : A \vdash \mathsf{POP3body}$. Defining $\mathsf{POP3} = \mathsf{port}?[x : A^1].\mathsf{POP3body}$ gives $\mathsf{port} : \widehat{\ }[A^1] \vdash \mathsf{POP3}$, which can be published as the specification of the server and its protocol.

The POP3 protocol permits an alternative authentication scheme, accessed by the apop command, in which the client sends a mailbox name and an au-

thenticating string simultaneously. This option does not have to be provided, but a server which does implement it requires a channel of type B, where B is obtained from A by adding a third option to the first choice:

$$\mathsf{apop} : ?[\mathsf{str}, \mathsf{str}] \,.\, \oplus\langle \mathsf{error} : ![\mathsf{str}] \,.\, X, \mathsf{ok} : ![\mathsf{str}] \,.\, T \rangle$$

A server which implements the apop command can be typed as follows: $\mathsf{port} : \hat{}[B^1] \vdash \mathsf{newPOP3}$. Now suppose that client is a POP3 client which does not know about the apop command. As before, we have $\mathsf{client} = (\nu x : A^2)\mathsf{port}![x].\mathsf{clientbody}$ where $x : A^1 \vdash \mathsf{clientbody}$. This gives $\mathsf{port} : \hat{}[A^1] \vdash \mathsf{client}$. The following derivation shows that client can also be typed in the environment $\mathsf{port} : \hat{}[B^1]$, and can therefore be put in parallel with newPOP3. The key fact is that $A \leqslant B$, because the top-level options provided by A are a subset of those provided by B.

$$\dfrac{\dfrac{x : \overline{A}^1 \vdash \mathsf{clientbody} \quad \hat{}[B] \leqslant ![A]}{\mathsf{port} : \hat{}[B^1], x : A^2 \vdash \mathsf{port}\,![x] \,.\, \mathsf{clientbody}}\,\text{T-Out}}{\mathsf{port} : \hat{}[B^1] \vdash (\nu x : A^2)\mathsf{port}\,![x] \,.\, \mathsf{clientbody}}\,\text{T-New}$$

Space does not permit us to present the definition of POP3body, but we claim that it is simpler and more readable than an equivalent definition in conventional π-calculus or Pict. The key factor is that the session type of x allows it to be used for all the messages exchanged in a complete POP3 session. Without session types, the client has to create a fresh channel every time it needs to send a message of a different type to the server; these channels also have to be sent to the server before use, which adds an overhead to every communication and therefore also to the channel types. Also in this case, the subtype relation on non-session types does not describe the relationship between interactions with POP3 and with newPOP3.

6 Conclusions and Future Work

We have defined a language whose type system incorporates session types, as suggested by Honda et al. [5,20], and subtyping, based on Pierce and Sangiorgi's work [15] and extended to session types. Session channels must be controlled linearly in order to guarantee that messages go to the correct destinations, and we have adapted the work of Kobayashi et al. [9] for this purpose. Our language differs minimally from the π-calculus, the only additions being primitives for offering and making labelled choices. Unlike Honda et al. we do not introduce special syntax for establishing and manipulating session channels; everything is taken care of by the typing rules. We have advocated using a session type as part of the published specification of a server's protocol, so that static type-checking can be used to verify that client implementations behave correctly. Using the POP3 protocol as an example, we have shown that subtyping increases the utility of this idea: if upgrading a server causes its protocol to have a session type which is a supertype of its original session type, then existing client implementations are still type-correct with respect to the new server.

Session types have some similarities to the types for active objects studied by Nierstrasz [14] and Puntigam [18,19]. Both incorporate the idea of a type which specifies a sequence of interactions. The main difference seems to be that in the case of active objects, types can specify interdependencies between interactions on different channels. However, the underlying framework (concurrent objects with method invocation, instead of channel-based communication between processes) is rather different, and we have not yet made a detailed comparison of the two systems.

The present paper is the first report of our work on a longer term project to investigate advanced type systems in the context of the Pict [16] programming language. Our next goal is to extend the Pict compiler to support the type system presented here. Because Pict is based on the *asynchronous* π-calculus [4,6,1] the output prefixing of our language will have to be encoded by explicitly attaching a continuation channel to each message. Initially we will work with a non-polymorphic version of Pict; later, after more theoretical study of the interplay between session types and polymorphism, we will integrate session types with the full Pict type system including polymorphism. The Pict compiler uses a powerful partial type inference technique [17], and it will be interesting to see how it can be extended to handle session types. Because of the value of explicit session types as specifications, we might not want to allow the programmer to omit them completely; however, automatic inference of, for example, some usage annotations will probably be very useful.

The implementation will allow us to gain more experience of programming with sessions, which in turn should suggest other typing features which can usefully be added to the system—for example, it would be interesting to consider Kobayashi's type system [7,8] for partial deadlock-freedom.

Acknowledgements

Malcolm Hole is funded by the EPSRC project "Novel Type Systems for Concurrent Programming Languages" (GR/L75177). Simon Gay is partially funded by the same EPSRC project, and also by a grant from the Nuffield Foundation. We thank the anonymous referees for their comments and suggestions. Paul Taylor's proof tree macros were used in the production of this paper.

References

1. G. Boudol. Asynchrony and the π-calculus (note). Rapporte de Recherche 1702, INRIA Sofia-Antipolis, May 1992.
2. S. J. Gay. A sort inference algorithm for the polyadic π-calculus. In *Proceedings, 20th ACM Symposium on Principles of Programming Languages*. ACM Press, 1993.
3. J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1–102, 1987.
4. K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *Proceedings of the European Conference on Object-Oriented Programming*, LNCS. Springer-Verlag, 1994.

5. K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *Proceedings of the European Symposium on Programming*, Lecture Notes in Computer Science. Springer-Verlag, 1998.
6. K. Honda and N. Yoshida. Combinatory representation of mobile processes. In *Proceedings, 21st ACM Symposium on Principles of Programming Languages*, 1994.
7. N. Kobayashi. A partially deadlock-free typed process calculus. In *Proceedings, Twelfth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1997.
8. N. Kobayashi. A partially deadlock-free typed process calculus. *ACM Transactions on Programming Languages and Systems*, 20:436–482, 1998.
9. N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. In *Proceedings, 23rd ACM Symposium on Principles of Programming Languages*, 1996.
10. I. Mackie. Lilac : A functional programming language based on linear logic. *Journal of Functional Programming*, 4(4):1–39, October 1994.
11. R. Milner. The polyadic π-calculus: A tutorial. Technical Report 91-180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1991.
12. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–77, September 1992.
13. J. Myers and M. Rose. Post office protocol version 3, May 1996. Internet Standards RFC1939.
14. O. Nierstrasz. Regular types for active objects. *ACM Sigplan Notices*, 28(10):1–15, October 1993.
15. B. Pierce and D. Sangiorgi. Types and subtypes for mobile processes. In *Proceedings, Eighth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1993.
16. B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. Technical Report CSCI 476, Computer Science Department, Indiana University, 1997. To appear in *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, MIT Press, 1998.
17. B. C. Pierce and D. N. Turner. Local type inference. In *Proceedings, 25th ACM Symposium on Principles of Programming Languages*, 1998.
18. F. Puntigam. Synchronization expressed in types of communication channels. In *Proceedings of the European Conference on Parallel Processing (Euro-Par'96)*, volume 1123 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
19. F. Puntigam. Coordination requirements expressed in types for active objects. In M. Aksit and S. Matsuoka, editors, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.
20. K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *Proceedings of the 6th European Conference on Parallel Languages and Architectures*, number 817 in Lecture Notes in Computer Science. Springer-Verlag, 1994.
21. D. N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis, University of Edinburgh, 1996.

Types for Safe Locking

Cormac Flanagan and Martín Abadi

Systems Research Center, Compaq
{flanagan|ma}@pa.dec.com

Abstract. A race condition is a situation where two threads manipulate a data structure simultaneously, without synchronization. Race conditions are common errors in multithreaded programming. They often lead to unintended nondeterminism and wrong results. Moreover, they are notoriously hard to diagnose, and attempts to eliminate them can introduce deadlocks. In practice, race conditions and deadlocks are often avoided through prudent programming discipline: protecting each shared data structure with a lock and imposing a partial order on lock acquisitions. In this paper we show that this discipline can be captured (if not completely, to a significant extent) through a set of static rules. We present these rules as a type system for a concurrent, imperative language. Although weaker than a full-blown program-verification calculus, the type system is effective and easy to apply. We emphasize a core, first-order type system focused on race conditions; we also consider extensions with polymorphism, existential types, and a partial order on lock types.

1 Races, Locks, and Types

Programming with multiple threads introduces a number of pitfalls, such as race conditions and deadlocks. A race condition is a situation where two threads manipulate a data structure simultaneously, without synchronization. Race conditions are common, insidious errors in multithreaded programming. They often lead to unintended nondeterminism and wrong results. Moreover, since race conditions are timing-dependent, they are notoriously hard to track down. Attempts to eliminate race conditions by using lock-based synchronization can introduce other errors, in particular deadlocks. A deadlock occurs when no thread can make progress because each is blocked on a lock held by some other thread.

In practice, both race conditions and deadlocks are often avoided through careful programming discipline [5]. Race conditions are avoided by protecting each shared data structure with a lock, and accessing the data structure only when the protecting lock is held. Deadlocks are avoided by imposing a strict partial order on locks and ensuring that each thread acquires locks only in increasing order. However, this programming discipline is not well supported by existing development tools. It is difficult to check if a program adheres to this discipline, and easy to write a program that does not by mistake. A single unprotected access in an otherwise correct program can produce a timing-dependent race condition whose cause may take weeks to identify.

S.D. Swierstra (Ed.): ESOP/ETAPS'99, LNCS 1576, pp. 91–108, 1999.
© Springer-Verlag Berlin Heidelberg 1999

In this paper we show that this programming discipline can be captured through a set of static rules. We present those rules as a type system for a concurrent, imperative language. We initially consider a first-order type system focused on race conditions. The type system supports dynamic thread creation and the dynamic allocation of locks and reference cells. We then consider extensions, such as universal and existential types, that increase the expressiveness of the system. We also outline an extension that eliminates deadlock by enforcing a strict partial order on lock acquisitions.

Since the programming discipline dictates that a thread can access a shared data structure only when holding a corresponding lock, our type systems provide rules for proving that a thread holds a given lock at a given program point. The rules rely on *singleton lock types*. A singleton lock type is the type of a single lock. Therefore, we can represent a lock l at the type level with the singleton lock type that contains l, and we can assert that a thread holds l by referring to that type rather than to the lock l. The type of a reference cell mentions both the type of the contents of the cell and the singleton lock type of the lock that protects the cell. Thus, singleton lock types provide a simple way of injecting lock values into the type level.

A set of singleton lock types forms a *permission*. During typechecking, each expression is analyzed in the context of a permission; including a singleton lock type in the permission amounts to assuming that the corresponding lock is held during evaluation of the expression. In addition, a permission decorates each function type and each function definition, representing the set of locks that must be held before a function call.

We study typechecking rather than type inference, so we do not show how to infer which lock protects a reference cell or which permission may decorate a function definition. We simply assume that the programmer can provide such information explicitly, and leave type inference as an open problem.

There is a significant body of previous work in this area, but most earlier approaches are either unsound (i.e., do not detect all race conditions) [22], deal only with finite state spaces [7,10,12], or do not handle mainstream shared-variable programming paradigms [1,16]. In contrast, we aim to give a sound type system for statically verifying the absence of race conditions in a programming language with shared variables. We defer a more detailed discussion of related work to section 7.

The next section describes a first-order type system for a concurrent, imperative language. Section 3 presents the operational semantics of the language, which is the basis for the race-freeness theorem of section 4. Section 5 extends the type system with universal and existential types. Section 6 further extends the type system in order to prevent deadlocks. Section 7 discusses related work. We conclude with section 8. For the sake of brevity, we omit proofs. Moreover, we give the type systems of sections 5 and 6 without corresponding operational semantics and correctness theorems. The operational semantics are straightforward. To date, we have studied how to extend the correctness theorem of section 4 to the type systems of section 5, but only partially to that of section 6.

$$
\begin{array}{rcl}
V \in \mathit{Value} & = & c \mid x \mid \lambda^p x{:}t.\, e \\
c \in \mathit{Const} & = & \mathit{unit} \\
x, y \in \mathit{Var} & & \\
e \in \mathit{Exp} & = & V \\
& \mid & e\ e \\
& \mid & \mathit{ref}_m e \mid\ !e \mid e := e \\
& \mid & \mathit{fork}\ e \\
& \mid & \mathit{new\text{-}lock}\ x{:}m\ \mathit{in}\ e \\
& \mid & \mathit{sync}\ e\ e
\end{array}
\qquad
\begin{array}{rcl}
s, t \in \mathit{Type} & = & \mathit{Unit} \\
& \mid & t \to^p t \\
& \mid & \mathit{Ref}_m t \\
& \mid & m \\
m, n, o \in \mathit{TypeVar} & & \\
p, q \in \mathit{Permission} & = & \mathcal{P}(\mathit{TypeVar})
\end{array}
$$

Fig. 1. A concurrent, imperative language.

2 First-order Types against Races

We start by considering a first-order type system focused on race conditions, and defer deadlock prevention to section 6. We formulate our type system for the concurrent, imperative language described in figure 1. The language is call-by-value, and includes values (constants, variables, and function definitions), applications, and the usual imperative operations on reference cells: allocation, dereferencing, and assignment. Although the language does not include explicit support for recursion, recursive functions can be encoded using reference cells, as described in section 2.2 below.

The language allows multithreaded programs by including the operation *fork* e which spawns a new thread for the evaluation of e. This evaluation is performed only for its effect; the result of e is never used. Locks are provided for thread synchronization. A lock has two states, locked and unlocked, and is initially unlocked. The expression *new-lock* $x : m$ *in* e dynamically allocates a new lock, binds x to that lock, and then evaluates e. It also introduces the type variable m which denotes the singleton lock type of the new lock. The expression *sync* e_1 e_2 is evaluated in a manner similar to Java's `synchronized` statement [14]: the subexpression e_1 is evaluated first, and should yield a lock, which is then acquired; the subexpression e_2 is then evaluated; and finally the lock is released. The result of e_2 is returned as the result of the *sync* expression. While evaluating e_2, the current thread is said to *hold* the lock, or, equivalently, is in a *critical section* on the lock. Any other thread that attempts to acquire the lock blocks until the lock is released. Locks are not reentrant; that is, a thread cannot reacquire a lock that it already holds. A new thread does not inherit locks held by its parent thread.

2.1 The Type Rules

The type of an expression depends on a *typing environment* E, which maps program variables to types, and maps type variables to the kind *Lock* (the kind

Judgments

$E \vdash \diamond$	E is a well-formed typing environment
$E \vdash t$	t is a well-formed type in E
$E \vdash p$	p is a well-formed permission in E
$E \vdash s <: t$	s is a subtype of t in E
$E \vdash p <: q$	p is a subpermission of q in E
$E\,;p \vdash e : t$	e is a well-typed expression of type t in E with p

Rules

$$\emptyset \vdash \diamond \quad \text{(Env } \emptyset\text{)}$$

$$\frac{E \vdash t \quad x \notin dom(E)}{E, x : t \vdash \diamond} \quad \text{(Env } x\text{)}$$

$$\frac{E \vdash \diamond \quad m \notin dom(E)}{E, m :: Lock \vdash \diamond} \quad \text{(Env } m\text{)}$$

$$\frac{E \vdash \diamond}{E \vdash Unit} \quad \text{(Type Unit)}$$

$$\frac{E \vdash s \quad E \vdash t \quad E \vdash p}{E \vdash s \to^p t} \quad \text{(Type Fun)}$$

$$\frac{E \vdash t \quad E \vdash m}{E \vdash Ref_m\, t} \quad \text{(Type Ref)}$$

$$\frac{E, m :: Lock, E' \vdash \diamond}{E, m :: Lock, E' \vdash m} \quad \text{(Type Lock)}$$

$$\frac{E \vdash \diamond \qquad E \vdash m \quad \text{for all } m \in p}{E \vdash p} \quad \text{(Perm)}$$

$$\frac{E \vdash p \quad E \vdash q \quad p \subseteq q}{E \vdash p <: q} \quad \text{(Subperm)}$$

$$\frac{E \vdash t}{E \vdash t <: t} \quad \text{(Sub Refl)}$$

$$\frac{E \vdash s_1 <: t_1 \qquad E \vdash t_2 <: s_2 \qquad E \vdash p <: q}{E \vdash (t_1 \to^p t_2) <: (s_1 \to^q s_2)} \quad \text{(Sub Fun)}$$

$$\frac{E\,;\emptyset \vdash unit : Unit}{} \quad \text{(Exp Unit)}$$

$$\frac{E, x : t, E' \vdash \diamond}{E, x : t, E'\,;\emptyset \vdash x : t} \quad \text{(Exp } x\text{)}$$

$$\frac{E, x : s\,;p \vdash e : t}{E\,;\emptyset \vdash \lambda^p x{:}s.\, e : s \to^p t} \quad \text{(Exp Fun)}$$

$$\frac{E\,;p \vdash e_1 : s \to^p t \qquad E\,;p \vdash e_2 : s}{E\,;p \vdash e_1\ e_2 : t} \quad \text{(Exp Appl)}$$

$$\frac{E \vdash m \quad E\,;p \vdash e : t}{E\,;p \vdash ref_m e : Ref_m t} \quad \text{(Exp Ref)}$$

$$\frac{E\,;p \vdash e : Ref_m t \quad m \in p}{E\,;p \vdash !e : t} \quad \text{(Exp Deref)}$$

$$\frac{E\,;p \vdash e_1 : Ref_m t \qquad E\,;p \vdash e_2 : t \quad m \in p}{E\,;p \vdash e_1 := e_2 : Unit} \quad \text{(Exp Set)}$$

$$\frac{E\,;\emptyset \vdash e : t}{E\,;\emptyset \vdash fork\, e : Unit} \quad \text{(Exp Fork)}$$

$$\frac{E, m :: Lock, x : m\,;p \vdash e : t \qquad E \vdash p \quad E \vdash t}{E\,;p \vdash new\text{-}lock\, x{:}m\ in\ e : t} \quad \text{(Exp Lock)}$$

$$\frac{E\,;p \vdash e_1 : m \qquad E\,;p \cup \{m\} \vdash e_2 : t}{E\,;p \vdash sync\ e_1\ e_2 : t} \quad \text{(Exp Sync)}$$

$$\frac{E\,;p \vdash e : t \qquad E \vdash p <: q \quad E \vdash t <: s}{E\,;q \vdash e : s} \quad \text{(Exp Sub)}$$

Fig. 2. The first-order type system.

of singleton lock types). The typing environment is organized as a sequence of bindings, and we use $\emptyset$ to denote the empty environment.

$$E ::= \emptyset \mid E, x : t \mid E, m :: Lock$$

We define the type system using six judgments. These judgments are described in figure 2, together with rules for reasoning about the judgments. The core of the type system is the set of rules for the judgment $E ; p \vdash e : t$ (read "e is a well-typed expression of type t in typing environment E with permission p"). Our intent is that, if this judgment holds, then e is race-free and yields values of type t, provided the current thread holds at least the locks described by p, and the free variables of e are given bindings consistent with the typing environment E.

The rule (Exp Fun) for functions $\lambda^p x : s.\, e$ checks that e is race-free given permission p, and then records this permission as part of the function's type: $s \rightarrow^p t$. The rule (Exp Appl) ensures that this permission is available at each call site of the function. The rule (Exp Ref) records the singleton lock type of the protecting lock as part of each reference-cell type: $Ref_m t$. The rules (Exp Deref) and (Exp Set) ensure that this lock is held (i.e., is in the current permission) whenever the reference cell is accessed. A single lock may protect several reference cells; in an obvious extension of our language, it could protect an entire record or object.

The rule (Exp Fork) typechecks the spawned expression using the empty permission, since threads never inherit locks from their parents. The rule (Exp Lock) for *new-lock* $x : m$ *in* e requires the type t of e to be well-formed in the original typing environment ($E \vdash t$). This requirement implies that t cannot contain the type variable m, and hence the *new-lock* expression cannot return the newly allocated lock. This constraint suffices to ensure that different singleton lock types of the same name are not confused. It is somewhat restrictive, but it can be circumvented with existential types, as described in section 5.2 below.

The rule (Exp Sync) for *sync* e_1 e_2 requires that e_1 yield a value of some singleton lock type m, and then typechecks e_2 with an extended permission that includes the type of the newly acquired lock. The use of this synchronization construct ensures that lock acquisition and release operations follow a stack-like discipline, which significantly simplifies the development of the type system.

The rule (Exp Sub) allows for subsumption on both types and permissions. If $E \vdash p <: q$, then any expression that is race-free with permission p is also race-free with the superset q of p.

2.2 Examples

For clarity, we present example programs using an extended language with integers, *let*-expressions, and a sequential composition operator (;). The program P_1 is a trivial example of using locks; it first allocates a lock and a reference cell protected by that lock, and then it acquires the lock and dereferences the cell. The program P_2 is slightly more complicated. It first allocates a lock and

defines a function g that increments reference cells protected by that lock. It then allocates two such reference cells, and uses g to increment both of them. The type of g is $((Ref_m\ Int) \rightarrow^{\{m\}} Int)$; it expresses that the protecting lock should be acquired before g is called.

$$
\begin{array}{lll}
P_1 \triangleq & \textit{new-lock}\ x\!:\!m\ \textit{in} & \\
& \quad \textit{let}\ y = \textit{ref}_m 1\ \textit{in} & \\
& \quad\quad \textit{sync}\ x\ !y &
\end{array}
\qquad
\begin{array}{lll}
P_2 \triangleq & \textit{new-lock}\ x\!:\!m\ \textit{in} & \\
& \quad \textit{let}\ g = \lambda^{\{m\}} z\!:\!\textit{Ref}_m \textit{Int}.\ z := !z + 1 & \\
& \quad\quad y_1 = \textit{ref}_m 1 & \\
& \quad\quad y_2 = \textit{ref}_m 2 & \\
& \quad \textit{in}\ \textit{sync}\ x\ (g\ y_1; g\ y_2) &
\end{array}
$$

Although the language does not include explicit support for recursion, we can encode recursion using reference cells. This idea is illustrated by the following program, which implements a server that repeatedly handles incoming requests. The core of this server is a recursive function that first allocates a new lock x_2 and associated reference cell y_2, then uses y_2 in handling an incoming request, and finally calls itself recursively to handle the next incoming request.

```
new-lock x1:m1 in                          ; Allocate a lock and a ref cell
  let y1 = ref_{m1}(λ^∅ x:Unit. x) in      ; initialized to the identity function.
    sync x1                                ; Acquire the lock and set the ref cell
      y1 := λ^∅ x:Unit.                    ; to the recursive function.
              new-lock x2:m2 in            ; Allocate a local lock
                let y2 = ref_{m2} 0 in     ; and a local ref cell.
                  ...                      ; Use the local ref cell.
                  (!y1 unit);              ; Call the recursive function.
      (!y1 unit)                           ; Start the server running.
```

The type variable m_2 denotes different singleton lock types at different stages of the execution, but these different types are not confused by the type system.

2.3 Expressiveness

Although the examples shown above are necessarily simple, the first-order type system is sufficiently expressive to verify a variety of non-trivial programs. In particular, any sequential program that is well-typed in the underlying sequential type system has an annotated variant that is well-typed in our type system. This annotated variant is obtained by enclosing the original program in the context $\textit{new-lock}\ x : m\ \textit{in}\ \textit{sync}\ x\ [\,]$ (which allocates and acquires a new lock of type m), annotating each reference cell with the type m, and annotating each function definition with the permission $\{m\}$. This approach can be generalized to multithreaded programs with a coarse locking strategy based on several global locks. It also suggests how to annotate thread-local data: writing $\textit{fork}\ (\textit{new-lock}\ x : m\ \textit{in}\ \textit{sync}\ x\ e)$ for spawning e and using x as the lock for protecting thread-local reference cells in e.

The type system does have some significant restrictions: functions cannot abstract over lock types, and the *new-lock* construct cannot return the newly

allocated lock; these restrictions are overcome in section 5. In addition, our type system is somewhat over-conservative in that it does not allow simultaneous reads of a data structure, even though simultaneous reads are not normally considered race conditions, and in fact many programs use reader-writer locks to permit such simultaneous reads [5]. We believe that adding a treatment of reader-writer locks to our type system should not be difficult.

3 Operational Semantics

We specify the operational semantics of our language using the abstract machine described in figure 3. The machine evaluates a program by stepping through a sequence of states. A state consists of three components: a lock store, a reference store, and a collection of expressions, each of which represents a thread. The expressions are written in a slight extension of the language *Exp*, called Exp_r, which includes the new construct *in-sync*. Since the result of the program is the result of its initial thread, the order of the threads in a state matters; therefore, we organize the threads as a sequence, and the first thread in the sequence is always the initial thread. We use the notation T_i to mean the ith element of a thread sequence T, where the initial thread is at index 0, and we use $T.T'$ to denote the concatenation of two sequences.

Reference cells are kept in a reference store σ, which maps reference locations to values. Locks are kept in a lock store π, which maps lock locations to either 0 or 1; $\pi(l) = 1$ when the lock l is held by some thread. Reference locations and lock locations are simply special kinds of variables that can be bound only by the respective stores. For each lock location l, we introduce a type variable o_l to denote the corresponding singleton lock type. A lock store that binds a lock location l also implicitly binds the corresponding type variable o_l with kind *Lock*; the only value of type o_l is l.

The evaluation of a program starts in an initial state with empty lock and reference stores and with a single thread. Evaluation then takes place according to the machine's transition rules. These rules specify the behavior of the various constructs in the language. The evaluation terminates once all threads have been reduced to values, in which case the value of the initial thread is returned as the result of the program. We use the notation $e[V/x]$ to denote the capture-free substitution of V for x in e, and use $\sigma[r \mapsto V]$ to denote the store that agrees with σ except at r, which is mapped to V.

The transition rules are mostly straightforward. The only unusual rules are the ones for lock creation and for *sync* expressions. To evaluate the expression *new-lock* $x : m$ *in* e, the transition rule (Trans Lock) allocates a new lock location l and replaces occurrences of x in e with l. The rule also replaces occurrences of m in e with the type variable o_l. To evaluate the expression *sync* l e, the transition rule (Trans Sync) acquires the lock l and yields the term *in-sync* l e. This term denotes that the lock l has been acquired and that the subexpression e is currently being evaluated. Since *in-sync* l [ ] is an evaluation context, subsequent transitions evaluate the subexpression e. Once this subexpression yields a value,

Evaluator

$$\begin{array}{rcl} \mathit{eval} & \subseteq & \mathit{Exp} \times \mathit{Value} \\ \mathit{eval}(e, V) & \iff & \langle \emptyset, \emptyset, e \rangle \longmapsto^* \langle \pi, \sigma, V.\mathit{unit}.\cdots.\mathit{unit} \rangle \end{array}$$

State space

$$\begin{array}{rcl} S \in \mathit{State} & = & \mathit{LockStore} \times \mathit{RefStore} \times \mathit{ThreadSeq} \\ \pi \in \mathit{LockStore} & = & \mathit{LockLoc} \rightharpoonup \{0, 1\} \\ \sigma \in \mathit{RefStore} & = & \mathit{RefLoc} \rightharpoonup \mathit{Value} \\ l \in \mathit{LockLoc} & \subset & \mathit{Var} \\ r \in \mathit{RefLoc} & \subset & \mathit{Var} \\ T \in \mathit{ThreadSeq} & = & \mathit{Exp}_r^* \\ f \in \mathit{Exp}_r & = & V \mid f\ e \mid V\ f \mid \mathit{ref}_m f \mid !f \mid f := e \mid r := f \\ & \mid & \mathit{fork}\ e \mid \mathit{new\text{-}lock}\ x\!:\!m\ \mathit{in}\ e \\ & \mid & \mathit{sync}\ f\ e \mid \mathit{in\text{-}sync}\ l\ f \end{array}$$

Evaluation contexts

$$\begin{array}{rcl} \mathcal{E} & = & [\,] \mid \mathcal{E}\ e \mid V\ \mathcal{E} \mid \mathit{ref}_m \mathcal{E} \mid !\mathcal{E} \mid \mathcal{E} := e \mid r := \mathcal{E} \\ & \mid & \mathit{sync}\ \mathcal{E}\ e \mid \mathit{in\text{-}sync}\ l\ \mathcal{E} \end{array}$$

Transition rules

$$\langle \pi, \sigma, T.\mathcal{E}[\ (\lambda^p x\!:\!t.\ e)\ V\ ].T' \rangle \longmapsto \langle \pi, \sigma, T.\mathcal{E}[\ e[V/x]\ ].T' \rangle \qquad \text{(Trans Appl)}$$

$$\langle \pi, \sigma, T.\mathcal{E}[\ \mathit{ref}_m V\ ].T' \rangle \longmapsto \langle \pi, \sigma[r \mapsto V], T.\mathcal{E}[\ r\ ].T' \rangle \quad \text{if } r \notin \mathit{dom}(\sigma) \qquad \text{(Trans Ref)}$$

$$\langle \pi, \sigma, T.\mathcal{E}[\ !r\ ].T' \rangle \longmapsto \langle \pi, \sigma, T.\mathcal{E}[\ V\ ].T' \rangle \quad \text{if } \sigma(r) = V \qquad \text{(Trans Deref)}$$

$$\langle \pi, \sigma, T.\mathcal{E}[\ r := V\ ].T' \rangle \longmapsto \langle \pi, \sigma[r \mapsto V], T.\mathcal{E}[\ \mathit{unit}\ ].T' \rangle \qquad \text{(Trans Set)}$$

$$\langle \pi, \sigma, T.\mathcal{E}[\ \mathit{new\text{-}lock}\ x\!:\!m\ \mathit{in}\ e\ ].T' \rangle \longmapsto \langle \pi[l \mapsto 0], \sigma, T.\mathcal{E}[\ e[l/x, o_l/m]\ ].T' \rangle \quad \text{if } l \notin \mathit{dom}(\pi) \qquad \text{(Trans Lock)}$$

$$\langle \pi[l \mapsto 0], \sigma, T.\mathcal{E}[\ \mathit{sync}\ l\ e\ ].T' \rangle \longmapsto \langle \pi[l \mapsto 1], \sigma, T.\mathcal{E}[\ \mathit{in\text{-}sync}\ l\ e\ ].T' \rangle \qquad \text{(Trans Sync)}$$

$$\langle \pi[l \mapsto 1], \sigma, T.\mathcal{E}[\ \mathit{in\text{-}sync}\ l\ V\ ].T' \rangle \longmapsto \langle \pi[l \mapsto 0], \sigma, T.\mathcal{E}[\ V\ ].T' \rangle \qquad \text{(Trans In-Sync)}$$

$$\langle \pi, \sigma, T.\mathcal{E}[\ \mathit{fork}\ e\ ].T' \rangle \longmapsto \langle \pi, \sigma, T.\mathcal{E}[\ \mathit{unit}\ ].T'.e \rangle \qquad \text{(Trans Fork)}$$

Fig. 3. The abstract machine.

Judgment

$\vdash S : t$ S is a well-typed state of type t

Rules

$$\frac{\begin{array}{c} dom(\pi) = \{l_1, \ldots, l_j\} \qquad dom(\sigma) = \{r_1, \ldots, r_k\} \\ E = o_{l_1} :: Lock,\, l_1 : o_{l_1}, \ldots, o_{l_j} :: Lock,\, l_j : o_{l_j},\, r_1 : Ref_{n_1}\, s_1, \ldots, r_k : Ref_{n_k}\, s_k \\ \forall i \in 1..k.\ E\,;\emptyset \vdash \sigma(r_i) : s_i \\ |T| > 0 \qquad \forall i < |T|.\ E\,;\emptyset \vdash T_i : t_i \end{array}}{\vdash \langle \pi, \sigma, T \rangle : t_0}$$

$$\frac{E\,;\emptyset \vdash l : m \qquad E\,;(p \cup \{m\}) \vdash f : t}{E\,;p \vdash \textit{in-sync}\ l\ f : t}$$

Fig. 4. Additional judgment and rules for typing states.

the transition rule (Trans In-Sync) releases the lock and returns that value as the result of the original *sync* expression. We say that an expression f is in a *critical section* on a lock location l if $f = \mathcal{E}[\ \textit{in-sync}\ l\ f'\ ]$ for some evaluation context $\mathcal{E}$ and expression f'.

The machine arbitrarily interleaves the execution of threads. Since different interleavings may yield different results, the evaluator *eval* is a proper relation and not simply a partial function.

We use the semantics to formalize the notion of a race condition. An expression f *accesses* a reference location r if there exists some evaluation context $\mathcal{E}$ such that $f = \mathcal{E}[\ !r\ ]$ or $f = \mathcal{E}[\ r := V\ ]$. A state has a *race condition* if its thread sequence contains two expressions that access the same reference location. A program e has a race condition if its evaluation may yield a state with a race condition, that is, if there exists a state S such that $\langle \emptyset, \emptyset, e \rangle \longmapsto^* S$ and S has a race condition.

4 Well-typed Programs Don't Have Races

The fundamental property of the type system is that well-typed programs do not have race conditions. The first component of the proof of this property is a subject reduction result stating that typing is preserved during evaluation. To prove this result, we extend typing judgments from expressions in *Exp* to expressions in Exp_r and then to machine states as shown in figure 4. The judgment $\vdash S : t$ says that S is a well-typed state yielding values of type t.

Lemma 1 (Subject Reduction). *If* $\vdash S : t$ *and* $S \longmapsto S'$, *then* $\vdash S' : t$.

Independently of the type system, locks provide mutual exclusion, in that two threads can never be in a critical section on the same lock. The judgment $\vdash_{cs} S$

Judgments

$\mathcal{M} \vdash_{cs} f$	f has exactly one critical section for each lock in $\mathcal{M}$
$\vdash_{cs} S$	S is well-formed with respect to critical sections

Rules

$$\frac{f = V \mid \mathit{fork}\ e \mid \mathit{new\text{-}lock}\ x{:}m\ \mathit{in}\ e}{\emptyset \vdash_{cs} f} \qquad \frac{\mathcal{M} \vdash_{cs} f}{\mathcal{M} \uplus \{l\} \vdash_{cs} \mathit{in\text{-}sync}\ l\ f}$$

$$\frac{\begin{array}{c}\mathcal{M} \vdash_{cs} f \\ f' = f\ e \mid V\ f \mid \mathit{ref}_m f \mid\ !f \\ \mid\ f := e \mid r := f \mid \mathit{sync}\ f\ e\end{array}}{\mathcal{M} \vdash_{cs} f'} \qquad \frac{\begin{array}{c}\forall i < |T|.\ \mathcal{M}_i \vdash_{cs} T_i \\ \mathcal{M} = \mathcal{M}_0 \uplus \cdots \uplus \mathcal{M}_{|T|-1} \\ \forall l \in \mathcal{M}.\ \pi(l) = 1\end{array}}{\vdash_{cs} \langle \pi, \sigma, T \rangle}$$

Fig. 5. Additional judgments and rules for reasoning about critical sections.

says that at most one thread is in a critical section on each lock in S (see figure 5). According to Lemma 2, the property $\vdash_{cs} S$ is maintained during evaluation.

Lemma 2 (Mutual Exclusion). *If* $\vdash_{cs} S$ *and* $S \longmapsto S'$, *then* $\vdash_{cs} S'$.

Lemma 3 says that a well-typed thread accesses a reference cell only when it holds the protecting lock.

Lemma 3. *Suppose that* $E\,;\emptyset \vdash f : t$, *and* f *accesses reference location* r. *Then* $E\,;\emptyset \vdash r : \mathit{Ref}_m s$ *for some lock type* m *and type* s. *Furthermore, there exists lock location* l *such that* $E\,;\emptyset \vdash l : m$ *and* f *is in a critical section on* l.

This lemma implies that states that are well-typed and well-formed with respect to critical sections do not have race conditions.

Lemma 4. *Suppose* $\vdash S : t$ *and* $\vdash_{cs} S$. *Then* S *does not have a race condition.*

We conclude that well-typed programs do not have race conditions.

Theorem 1. *If* $\emptyset\,;\emptyset \vdash e : t$ *then* e *does not have a race condition.*

5 Second-order Types against Races

Although the first-order type system of section 2 is applicable to a variety of multithreaded programs, there are many race-free programs that it cannot verify. This section describes extensions that allow the verification of more complex programs. These extensions rely on polymorphism and type abstraction, which are fairly easy to incorporate into a type-based approach such as ours.

Extended syntax

$$\begin{array}{rcl} V \in \mathit{Value} & = & \ldots \mid \Lambda m :: \mathit{Lock}.\, V \\ e \in \mathit{Exp} & = & \ldots \mid e[m] \\ s, t \in \mathit{Type} & = & \ldots \mid \forall m :: \mathit{Lock}.\, t \end{array}$$

Additional rules

$$\frac{E, m :: \mathit{Lock} \vdash t}{E \vdash \forall m :: \mathit{Lock}.\, t} \qquad \frac{E, m :: \mathit{Lock}\,; \emptyset \vdash V : t}{E\,; \emptyset \vdash \Lambda m :: \mathit{Lock}.\, V : (\forall m :: \mathit{Lock}.\, t)}$$

$$\frac{E, m :: \mathit{Lock} \vdash s <: t}{E \vdash (\forall m :: \mathit{Lock}.\, s) <: (\forall m :: \mathit{Lock}.\, t)} \qquad \frac{E\,; p \vdash e : (\forall m :: \mathit{Lock}.\, t) \qquad E \vdash n}{E\,; p \vdash e[n] : t[n/m]}$$

Fig. 6. Extending the first-order type system with universal types.

5.1 Polymorphism over Lock Types

The first-order type system does not permit functions parameterized by lock types. To overcome this limitation, we extend the type system to include polymorphism, as described in figure 6. The only unusual aspect of this extension is that we require the body of a polymorphic abstraction to be a value. This restriction avoids the need to annotate polymorphic abstractions with permissions, since the trivial evaluation of a value requires only the empty permission. (If needed, we can still include a non-value expression in a polymorphic abstraction by wrapping the expression in a function definition.)

Examples The following program P_3 defines a polymorphic function for incrementing a reference cell. The function abstracts over both the reference cell and the type of the lock protecting the reference cell, and the caller is responsible for acquiring that lock. The program P_4 is similar, except that the lock is acquired inside the increment function.

$$\begin{array}{rcl} P_3 & \triangleq & \mathit{let}\ g = \Lambda n :: \mathit{Lock}. \\ & & \quad \lambda^{\{n\}} z : \mathit{Ref}_n \mathit{Int}. \\ & & \quad\quad z :=\!!z + 1 \\ & & \mathit{in}\ \mathit{new\text{-}lock}\, x : m\ \mathit{in} \\ & & \quad \mathit{let}\ y = \mathit{ref}_m 0\ \mathit{in} \\ & & \quad\quad \mathit{sync}\ x\ (g[m]\ y) \end{array}$$

$$\begin{array}{rcl} P_4 & \triangleq & \mathit{let}\ g = \Lambda n :: \mathit{Lock}. \\ & & \quad \lambda^{\emptyset} w : n. \\ & & \quad\quad \lambda^{\emptyset} z : \mathit{Ref}_n \mathit{Int}. \\ & & \quad\quad\quad \mathit{sync}\ w\ (z :=\!!z + 1) \\ & & \mathit{in}\ \mathit{new\text{-}lock}\, x : m\ \mathit{in} \\ & & \quad \mathit{let}\ y = \mathit{ref}_m 0\ \mathit{in} \\ & & \quad\quad g[m]\ x\ y \end{array}$$

5.2 Existential Quantification over Lock Types

All our type systems require that the result type of *new-lock* $x : m$ *in* e be a well-formed type in the environment of the *new-lock* expression. This requirement forbids returning the type variable m out of the scope of its binding *new-lock* expression, and hence unfortunately excludes some useful programming patterns.

Extended syntax

$$\begin{array}{rcl} V \in \mathit{Value} & = \ldots & \mid \mathit{pack}\, m :: \mathit{Lock} = n\ \mathit{with}\ V \\ e \in \mathit{Exp} & = \ldots & \mid \mathit{open}\, e\ \mathit{as}\, m :: \mathit{Lock}, x : t\ \mathit{in}\ e \\ s, t \in \mathit{Type} & = \ldots & \mid \exists m :: \mathit{Lock}.\, t \end{array}$$

Additional rules

$$\frac{E, m :: \mathit{Lock} \vdash t}{E \vdash \exists m :: \mathit{Lock}.\, t} \qquad \frac{E, m :: \mathit{Lock} \vdash s <: t}{E \vdash (\exists m :: \mathit{Lock}.\, s) <: (\exists m :: \mathit{Lock}.\, t)}$$

$$\frac{E \vdash n \qquad E\,; \emptyset \vdash V[n/m] : t[n/m]}{E\,; \emptyset \vdash \mathit{pack}\, m :: \mathit{Lock} = n\ \mathit{with}\ V : (\exists m :: \mathit{Lock}.\, t)}$$

$$\frac{\begin{array}{c} E\,; p \vdash e_1 : (\exists m :: \mathit{Lock}.\, t) \\ E, m :: \mathit{Lock}, x : t\,; p \vdash e_2 : s \\ E \vdash s \end{array}}{E\,; p \vdash \mathit{open}\, e_1\ \mathit{as}\, m :: \mathit{Lock}, x : t\ \mathit{in}\ e_2 : s}$$

Fig. 7. Extending the first-order type system with existential types.

For example, consider a multithreaded implementation of binary trees. To reduce lock contention, the implementation may protect each node with a separate lock. The node allocation routine thus needs to create a fresh lock, say of type m, and return a new node of type $m \times \mathit{Ref}_m \alpha \times \mathit{Ref}_m \alpha$, where α is the type of the node's children. But including m in the return type implies lifting it out of its binding *new-lock* expression, and is forbidden by the type system.

We circumvent this restriction by noting that the caller of the allocation routine does not care which singleton lock type is used to protect the node. The caller requires only that there exist some lock type m such that the node has type $m \times \mathit{Ref}_m \alpha \times \mathit{Ref}_m \alpha$. This insight suggests the use of existential types for typing such programs. It is straightforward to extend the type system with existential types, as outlined in figure 7. The type rules closely follow the conventional rules for existential types [6]. In the rule for *pack*, the lock type n is hidden and replaced with the type variable m in the resulting existential type $(\exists m :: \mathit{Lock}.\, t)$. We do not explicitly allow for renaming in the rule for *open*, since renaming can be accomplished using α-conversion, if necessary.

Examples Some of the following examples use product, sum, and recursive types, which are easily added to the type system, as in [6]. Values of these additional types are manipulated by the following operations: $\langle e_1, \ldots, e_n \rangle$ creates a value of a product type, whose components are retrieved by the operations *first*, *second*, etc.; *inLeft* creates a value of a sum type, whose component is retrieved by *asLeft*; *inRight* and *asRight* behave in a similar manner; and *fold* and *unfold* convert between a recursive type $\mu\alpha.\, t$ and its unfolding $t[(\mu\alpha.\, t)/\alpha]$.

The following expression P_5 provides a simple example of using existential types. This expression has type $\exists m :: Lock.\ (m \times Ref_m Int)$, and it returns a pair consisting of a lock and a reference cell protected by that lock. The expression P_6 opens P_5 and retrieves the value of the reference cell.

$$\begin{array}{l} P_5 \triangleq \ new\text{-}lock\ x : n\ in \\ \quad let\ y = \langle x, (ref_n\, 0)\rangle\ in \\ \quad\quad pack\ m :: Lock = n\ with\ y \end{array} \qquad \begin{array}{l} P_6 \triangleq \ open\ P_5\ as \\ \quad m :: Lock, y : (m \times Ref_m Int)\ in \\ \quad\quad sync\ first(y)\ !second(y) \end{array}$$

For a more realistic example, we reconsider how to implement binary trees using a separate lock to protect each node. In this implementation, a leaf node is represented as *unit*, and an interior node is represented as a triple, using an existential type to hide the type of the protecting lock. The type of a binary tree is thus:

$$T \triangleq \mu\alpha.\, (Unit + \exists m :: Lock.\, (m \times Ref_m \alpha \times Ref_m \alpha))$$

Some typical routines for manipulating binary trees are:

$$\begin{array}{lll} leaf & : T & \triangleq fold(inLeft(unit)) \\ \\ alloc\text{-}node & : T \to^{\emptyset} (T \to^{\emptyset} T) & \triangleq \lambda^{\emptyset} l : T. \\ & & \quad \lambda^{\emptyset} r : T. \\ & & \quad\quad new\text{-}lock\ x : n\ in \\ & & \quad\quad\quad pack\ m :: Lock = n\ with \\ & & \quad\quad\quad\quad fold(inRight(\langle x, ref_n l, ref_n r\rangle)) \\ \\ left\text{-}child & : T \to^{\emptyset} T & \triangleq \lambda^{\emptyset} x : T. \\ & & \quad open\ asRight(unfold(x)) \\ & & \quad\quad as\ m :: Lock, y : (m \times Ref_m T \times Ref_m T) \\ & & \quad\quad in\ sync\ first(y)\ !second(y) \end{array}$$

6 Preventing Deadlocks

The type systems described so far ensure that well-typed programs do not have race conditions. However, these programs may still suffer from other errors, in particular deadlocks.

We formalize the notion of deadlock using the operational semantics of figure 3. An expression f *requests* a lock l if $f = \mathcal{E}[\ sync\ l\ e\ ]$. A state is *deadlocked* if there is a cycle of threads in the state such that each thread requests a lock held by the next thread in the cycle. More precisely, a state $S = \langle \pi, \sigma, T\rangle$ is deadlocked if there exist lock locations $l_0, \ldots, l_n$ and indices $d_0, \ldots, d_{n-1}$ of T such that $n > 0$, $l_0 = l_n$, and for each $0 \le i < n$, thread T_{d_i} is in a critical section on l_i and requests lock l_{i+1}. A program e may deadlock if its evaluation may yield a deadlocked state, that is, if there exists a deadlocked state S such that $\langle \emptyset, \emptyset, e\rangle \longmapsto^* S$.

Judgments (in addition to those of figure 2)

$E \vdash m :: (\overline{m}_1, \overline{m}_2)$	m is in the interval $(\overline{m}_1, \overline{m}_2)$ in E
$E \vdash m_1 \prec m_2$	m_1 is less than m_2 in E
$E \vdash (\overline{m}_1, \overline{m}_2)$	$(\overline{m}_1, \overline{m}_2)$ is a well-formed, non-empty interval in E

Rules (partial list)

$$\frac{E \vdash \diamond \qquad m \notin dom(E) \qquad E \vdash (\overline{m}_1, \overline{m}_2)}{E, m :: (\overline{m}_1, \overline{m}_2) \vdash \diamond}$$

$$\frac{E \vdash \diamond \qquad \forall m_1 \in \overline{m}_1.\ \forall m_2 \in \overline{m}_2.\ E \vdash m_1 \prec m_2}{E \vdash (\overline{m}_1, \overline{m}_2)}$$

$$\frac{E, m :: (\overline{m}_1, \overline{m}_2), E' \vdash \diamond}{E, m :: (\overline{m}_1, \overline{m}_2), E' \vdash m :: (\overline{m}_1, \overline{m}_2)}$$

$$\frac{\overline{m}_1 \subseteq \overline{m}_2 \qquad E \vdash L_2 \prec L_1 \text{ or } L_1 = L_2}{E \vdash \langle \overline{m}_1, L_1 \rangle <: \langle \overline{m}_2, L_2 \rangle}$$

$$\frac{E \vdash \diamond \quad E \vdash m :: (\overline{m}_1, \overline{m}_2) \quad m_1 \in \overline{m}_1}{E \vdash m_1 \prec m}$$

$$\frac{E\,; \langle \overline{m}, L \rangle \vdash e : \mathit{Ref}_m t \qquad m \in \overline{m}}{E\,; \langle \overline{m}, L \rangle \vdash !e : t}$$

$$\frac{E \vdash \diamond \quad E \vdash m :: (\overline{m}_1, \overline{m}_2) \quad m_2 \in \overline{m}_2}{E \vdash m \prec m_2}$$

$$\frac{E\,; \langle \overline{m}, L \rangle \vdash e_1 : \mathit{Ref}_m t \qquad E\,; \langle \overline{m}, L \rangle \vdash e_2 : t \qquad m \in \overline{m}}{E\,; \langle \overline{m}, L \rangle \vdash e_1 := e_2 : \mathit{Unit}}$$

$$\frac{E \vdash \diamond \quad E \vdash m}{E \vdash m \prec \top}$$

$$\frac{E\,; \langle \emptyset, \bot \rangle \vdash e : t}{E\,; \langle \emptyset, \top \rangle \vdash \mathit{fork}\ e : \mathit{Unit}}$$

$$\frac{E \vdash \diamond \quad E \vdash m}{E \vdash \bot \prec m}$$

$$\frac{E, m :: (\overline{m}_1, \overline{m}_2), x : m\,; p \vdash e : t \qquad E \vdash p \quad E \vdash t}{E\,; p \vdash \mathit{new\text{-}lock}\ x : m :: (\overline{m}_1, \overline{m}_2)\ \mathit{in}\ e : t}$$

$$\frac{E \vdash m_1 \prec m \qquad E \vdash m \prec m_2}{E \vdash m_1 \prec m_2}$$

$$\frac{E\,; \langle \overline{m}, L \rangle \vdash e_1 : m \qquad E \vdash L \prec m \qquad E\,; \langle \overline{m} \cup \{m\}, m \rangle \vdash e_2 : t}{E\,; \langle \overline{m}, L \rangle \vdash \mathit{sync}\ e_1\ e_2 : t}$$

Fig. 8. Extending the type system for deadlock elimination (highlights).

In practice, deadlocks are commonly avoided by imposing a strict partial order on locks, and respecting this order when acquiring locks [5]. We capture this discipline by embodying it in an extension of our type system.

Our extended type system relies on annotations that specify a lock ordering. Whenever we introduce a singleton lock type, we must specify an appropriate order between that lock type and the other lock types in the program. If $\overline{m}_1$ and $\overline{m}_2$ are sets of lock types, then we use the notation $m :: (\overline{m}_1, \overline{m}_2)$ to mean that the lock type m is greater than each lock in $\overline{m}_1$ and is less than each lock in $\overline{m}_2$. Thus the interval $(\overline{m}_1, \overline{m}_2)$ specifies a kind. In the extended language, we use kinds of the form $(\overline{m}_1, \overline{m}_2)$ instead of the kind *Lock*:

$$\begin{array}{rcl|l} V \in \mathit{Value} & = \ldots & | & \Lambda m :: (\overline{m}_1, \overline{m}_2).\, V \\ & & | & \mathit{pack}\ m :: (\overline{m}_1, \overline{m}_2) = n\ \mathit{with}\ V \\ e \in \mathit{Exp} & = \ldots & | & \mathit{new\text{-}lock}\ x : m :: (\overline{m}_1, \overline{m}_2)\ \mathit{in}\ e \\ & & | & \mathit{open}\ e\ \mathit{as}\ m :: (\overline{m}_1, \overline{m}_2), x : t\ \mathit{in}\ e \\ s, t \in \mathit{Type} & = \ldots & | & \exists m :: (\overline{m}_1, \overline{m}_2).\, t \\ & & | & \forall m :: (\overline{m}_1, \overline{m}_2).\, t \end{array}$$

The type system ensures that locks are acquired in the appropriate order using the notion of a *locking level*. A locking level L is either a particular lock type, in which case any greater lock can be acquired, or $\bot$, in which case all locks can be acquired, or $\top$, in which case no lock can be acquired. We extend permissions to include a locking-level component, so a permission is a pair of a lock set and a locking level. The trivial, empty permission is $\langle \emptyset, \top \rangle$, and the initial permission (of forked threads and of the main program) is $\langle \emptyset, \bot \rangle$.

We extend the typing environment E to map type variables to intervals $(\overline{m}_1, \overline{m}_2)$. The judgment $E \vdash m_1 \prec m_2$ expresses that m_1 is less than m_2; the judgment $E \vdash (\overline{m}_1, \overline{m}_2)$ expresses that $(\overline{m}_1, \overline{m}_2)$ is a well-formed, non-empty interval; and the judgment $E \vdash m :: (\overline{m}_1, \overline{m}_2)$ expresses that the lock type m is in the interval $(\overline{m}_1, \overline{m}_2)$. In a well-formed environment, the ordering constraints on lock variables induce a strict partial order.

The necessary modifications to the type rules are outlined in figure 8. Most of the rules are straightforward adaptations of earlier rules. The rule for *fork* initializes a newly spawned thread with the locking level $\bot$, since that thread is free to acquire any lock. The rule for *sync* ensures that locks are acquired in increasing order. Collectively, the type rules check that threads respect the strict partial order on locks, as required by the discipline for preventing deadlocks.

7 Related Work

Race conditions and deadlocks have been studied for decades, for example in the program-verification literature. In this section, we mention some of the work most closely related to ours.

Warlock [22] is a system for detecting race conditions and deadlocks statically. Its goals are similar to those of our type system. The major differences are that Warlock is an implemented system applicable to substantial programs, and that

Warlock may fail to detect certain race conditions. This unsoundness is partly due to the target language (ANSI C), and partly due to two other difficulties that are overcome by our system. First, Warlock works by tracing execution paths, but it fails to trace paths through loops or recursive function calls. Second, Warlock appears to merge different locks of the same type, and so may fail to detect inconsistent locking.

Aiken and Gay [2] also investigate static race detection, in the somewhat different setting of SPMD programs. They present a system that has been used successfully on a variety of SPMD programs. Synchronization in these programs is performed using barriers. Since a barrier is a global operation not associated with any particular variable in the program, they do not develop machinery for tracking the association between reference cells and their protecting locks.

A number of analyses have been developed for concurrent languages such as CML [20]. Nielson and Nielson [19] present an analysis that predicts process and channel utilization and uses this information for optimization. Their analysis is based on the notion of behaviors, which are similar to our permissions. Colby [9] also presents an analysis that infers information about channel usage. Neither work treats race conditions or deadlocks.

Kobayashi [16] presents a first-order type system for a process calculus. His type system has a deadlock-free subset, and uses the notion of time tags, which are similar to our locking levels. Although Kobayashi considers some sophisticated determinism properties, he does not address race conditions directly.

Abramsky, Gay, and Nagarajan [1] present another type-based technique for avoiding deadlocks. Their work is based on interaction categories inspired by linear logic. It emphasizes issues of type structure, rather than their application to a specific programming language.

Dwyer and Clarke [11] describe a data-flow analysis for verifying certain correctness properties of concurrent programs, for example mutual exclusion on particular resources. The authors suggest that their analysis is not well suited for detecting global properties such as deadlock. Avrunin *et al.* [4] describe a toolset for analyzing concurrent programs. This toolset has been used for detecting race conditions and deadlocks in a variety of benchmarks, on a case-by-case basis.

Savage *et al.* [21] describe Eraser, a tool for detecting race conditions and deadlocks dynamically (rather than statically, as in our method). Although quite effective, Eraser may fail to detect certain race conditions and deadlocks because of insufficient test coverage. In general, static checking and testing are complementary, and they should both be used in the development of reliable software. Hybrid approaches (like that of the Cilk Determinator [8]) seem promising.

There is a large amount of work on model-checking of concurrent programs, particularly focused on finite-state systems (*e.g.*, [7,10,12]). Recently, Godefroid has applied model-checking techniques to C programs [13]; his approach, stateless state-space exploration, relies on dynamic observation rather than static analysis.

The permissions that we use are similar to effects [15,17,18] in that the permission of an expression constrains the effects that it may produce. Much work

has been done on effect reconstruction [3,23,24,25]. It may be possible to adapt these inference methods in our setting in order to remove the need for explicit lock annotations.

8 Conclusions

This paper describes how a type system can be used for avoiding two major pitfalls of multithreaded programming, namely race conditions and deadlocks. Our approach requires annotating programs with locking information. We believe that this information is usually known to competent programmers and often implicit in documentation. However, it may be worthwhile to investigate algorithms for inferring the annotations. Such algorithms would be helpful in tackling larger examples and in extending our techniques to programming languages more realistic than the one treated in this paper. Also helpful would be a mechanism for escaping from our type system when it proves too restrictive. We leave those issues for further work.

For sequential languages, standard type systems provide a means for expressing and checking fundamental correctness properties. We hope that type systems such as ours will play a similar role in the realm of multithreaded programming.

Acknowledgments

Comments from Mike Burrows, Rustan Leino, and Mark Lillibridge were helpful for this work.

References

1. S. Abramsky, S. Gay, and R. Nagarajan. A type-theoretic approach to deadlock-freedom of asynchronous systems. In *Theoretical Aspects of Computer Software*, volume 1281 of *Lecture Notes in Computer Science*, pages 295–320. Springer-Verlag, 1997.
2. A. Aiken and D. Gay. Barrier inference. In *Proceedings of the 25th Symposium on Principles of Programming Languages*, pages 243–354, 1998.
3. T. Amtoft, F. Nielson, and H. R. Nielson. Type and behaviour reconstruction for higher-order concurrent programs. *Journal of Functional Programming*, 7(3):321–347, 1997.
4. G. S. Avrunin, U. A. Buy, J. C. Corbett, L. K. Dillon, and J. C. Wileden. Automated analysis of concurrent systems with the constrained expression toolset. *IEEE Transactions on Software Engineering*, 17(11):1204–1222, 1991.
5. A. D. Birrell. An introduction to programming with threads. Research Report 35, Digital Equipment Corporation Systems Research Center, 1989.
6. L. Cardelli. Type systems. *Handbook of Computer Science and Engineering*, pages 2208–2236, 1997.
7. A. T. Chamillard, L. A. Clarke, and G. S. Avrunin. An empirical comparison of static concurrency analysis techniques. Technical Report 96-084, Department of Computer Science, University of Massachusetts at Amherst, 1996.

8. G.-I. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark. Detecting data races in Cilk programs that use locks. In *Proceedings of the 10th Symposium on Parallel Algorithms and Architectures*, pages 298–309, 1998.
9. C. Colby. Analyzing the communication topology of concurrent programs. In *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 202–213, 1995.
10. J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3):161–180, 1996.
11. M. B. Dwyer and L. A. Clarke. Data flow analysis for verifying properties of concurrent programs. Technical Report 94-045, Department of Computer Science, University of Massachusetts at Amherst, 1994.
12. L. Fajstrup, E. Goubault, and M. Raussen. Detecting deadlocks in concurrent systems. In *CONCUR'98: Concurrency Theory*, volume 1466 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
13. P. Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th Symposium on Principles of Programming Languages*, pages 174–186, 1997.
14. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
15. P. Jouvelot and D. Gifford. Algebraic reconstruction of types and effects. In *Proceedings of the 18th Symposium on Principles of Programming Languages*, pages 303–310, 1991.
16. N. Kobayashi. A partially deadlock-free typed process calculus. In *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science*, pages 128–139, 1997.
17. J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 47–57, 1988.
18. F. Nielson. Annotated type and effect systems. *ACM Computing Surveys*, 28(2):344–345, 1996. Invited position statement for the Symposium on Models of Programming Languages and Computation.
19. H. R. Nielson and F. Nielson. Higher-order concurrent programs with finite communication topology. In *Proceedings of the 21st Symposium on Principles of Programming Languages*, pages 84–97, 1994.
20. J. H. Reppy. CML: a higher-order concurrent language. In *ACM '91 Conference on Programming Language Design and Implementation.*, pages 293–305, 1991.
21. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.
22. N. Sterling. Warlock: A static data race analysis tool. In *USENIX Winter Technical Conference*, pages 97–106, 1993.
23. J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, 1992.
24. M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *Proceedings of the 21st Symposium on Principles of Programming Languages*, pages 188–201, 1994.
25. M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

Constructor Subtyping

Gilles Barthe[1,2] and Maria João Frade[1]

[1] Departamento de Informática, Universidade do Minho, Braga, Portugal
[2] Institutionen för Datavetenskap, Chalmers Tekniska Högskola, Göteborg, Sweden
{gilles,mjf}@di.uminho.pt

Abstract. *Constructor subtyping* is a form of subtyping in which an inductive type σ is viewed as a subtype of another inductive type τ if τ has more constructors than σ. As suggested in [5,12], its (potential) uses include proof assistants and functional programming languages.
In this paper, we introduce and study the properties of a simply typed λ-calculus with record types and datatypes, and which supports record subtyping and constructor subtyping. In the first part of the paper, we show that the calculus is confluent and strongly normalizing. In the second part of the paper, we show that the calculus admits a well-behaved theory of canonical inhabitants, provided one adopts expansive extensionality rules, including η-expansion, surjective pairing, and a suitable expansion rule for datatypes. Finally, in the third part of the paper, we extend our calculus with unbounded recursion and show that confluence is preserved.

1 Introduction

Type systems [3,8] lie at the core of modern functional programming languages, such as Haskell [28] or ML [26], and proof assistants, such as Coq [4] or PVS [32]. In order to improve the usability of these languages, it is important to devise flexible (and safe) type systems, in which programs and proofs may be written easily. A basic mechanism to enhance the flexibility of type systems is to endorse the set of types with a *subtyping* relation $\leq$ and to enforce a *subsumption* rule

$$\frac{a : A \qquad A \leq B}{a : B}$$

This basic mechanism of subtyping is powerful enough to capture a variety of concepts in computer science, see e.g. [9], and its use is spreading both in functional programming languages, see e.g. [25,30,31], and in proof assistants, see e.g. [7,24,32].

Constructor subtyping is a basic form of subtyping, suggested in [12] and developed in [5], in which an inductive type σ is viewed as a subtype of another inductive type τ if τ has more constructors than σ. As such, constructor subtyping captures in a type-theoretic context the ubiquitous use of subtyping as inclusion between inductively defined sets. In its simplest instance, constructor subtyping enforces subtyping from odd or even numbers to naturals, as illustrated in the following example, which introduces in a ML-like syntax the mutually recursive datatypes `Odd` and `Even`, and the `Nat` datatype:

S.D. Swierstra (Ed.): ESOP/ETAPS'99, LNCS 1576, pp. 109–127, 1999.
© Springer-Verlag Berlin Heidelberg 1999

```
datatype Odd  = s of Even        datatype Nat = 0
and      Even = 0                             | s of Nat
              | s of Odd ;                    | s of Odd
                                              | s of Even ;
```

Here `Even` and `Odd` are subtypes of `Nat` (i.e. `Even` $\leq$ `Nat` and `Odd` $\leq$ `Nat`), since every constructor of `Even` and `Odd` is also a constructor of `Nat`.

In a previous paper [5], the first author introduced and studied constructor subtyping for one first-order mutually recursive parametric datatype, and showed the calculus to be confluent and strongly normalizing. In the present paper, we improve on this work in several directions:

1. we extend constructor subtyping to the class of strictly positive, mutually recursive and parametric datatypes. In addition, the present calculus supports incremental definitions;
2. following recent trends in the design of proof assistants (and a well-established trend in the design of functional programming languages), we replace the elimination constructors of [5] by case-expressions. This leads to a simpler system, which is easier to use;
3. we define a set of expansive extensionality rules, including η-expansion, surjective pairing, and a suitable expansion rule for datatypes, so as to obtain a well-behaved theory of canonical inhabitants (i.e. of closed expressions in normal forms). The latter is fundamental for a proper semantical understanding of the calculus and for several applications related to proof assistants, such as unification.

The main technical contribution of this paper is to show that the calculus enjoys several fundamental meta-theoretical properties including confluence, subject reduction, strong normalization and a well-behaved theory of canonical inhabitants. These results lay the foundations for constructor subtyping and open the possibility of using constructor subtyping in programming languages and proof assistants, see Section 7.

Organization of the paper The paper is organized as follows: in Section 2, we provide an informal account of constructor subtyping. In Section 3, we introduce a simply typed λ-calculus with record types and datatypes, and which supports both record subtyping and constructor subtyping. In Section 4, we establish some fundamental meta-theoretical properties of the calculus. In Section 5, we motivate the use of expansive extensionality rules, show that they preserve confluence and strong normalization and lead to a well-behaved theory of canonical inhabitants. In Section 6, we extend our core language with fixpoint operators, and show the resulting calculus to be confluent. Finally, we conclude in Section 7. Because of space constraints, proofs are merely sketched or omitted. We refer the reader to [6] for further details.

Acknowledgments We are grateful to T. Altenkirch, P. Dybjer and L. Pinto for useful discussions on constructor subtyping. The first author is partially

supported by a TMR fellowship. The second author is partially supported by the LOGCOMP project.

2 An informal account of constructor subtyping

Constructor subtyping formalizes the view that an inductively defined set σ is a subtype of an inductively defined set τ if τ has more constructors than σ. As may be seen from the example of even, odd and natural numbers, the relative generality of constructor subtyping relies on the possibility for constructors to be overloaded and, to a lesser extent, on the possibility for datatypes to be defined in terms of previously introduced datatypes. The following example, which introduces the parametric datatypes `List` of lists and `NeList` of non-empty lists, provides further evidence.

```
datatype 'a List = nil
                 | cons of ('a * 'a List) ;

datatype 'a NeList = cons of ('a * 'a List) ;
```

Here `'a NeList` $\leq$ `'a List` since the only constructor of `'a NeList`, `cons : ('a * 'a List) →'a NeList` is matched by the constructor of `'a List`, `cons : ('a * 'a List) →'a List`.

The above examples reveal a possible pattern of constructor subtyping: for two parametric datatypes d and d' with the same arity, we set $d \leq d'$ if every declaration (`c` in case of a constant, `c of B` otherwise) of d is matched in d'.[1] Another pattern, used in [5], is to take subtyping as a primitive. Here we allow for the subtyping relation to be specified directly in the definition of the datatype. As shown below, such a pattern yields simpler definitions, with less declarations.

```
datatype Odd  = s of Even        datatype Nat  =  s of Nat
and      Even = 0                with     Odd ≤ Nat,
              | s of Odd ;                Even ≤ Nat ;
```

The original datatype may be recovered by adding a declaration of the form $c : \boldsymbol{\sigma} \rightarrow d'$ whenever $c : \boldsymbol{\sigma} \rightarrow d$ and $d \leq d'$. The same technique can be used to define `'a List` and `'a NeList`:

```
datatype 'a List = nil
and      'a NeList = cons of ('a * 'a List)
with     'a NeList ≤ 'a List ;
```

For the clarity of the exposition, we shall adopt the second pattern in examples, whereas we consider the first pattern in the formal definition of $\lambda_{\rightarrow,[],\mathsf{data}}$.

[1] For the sake of simplicity, we gloss over renamings and assume the parameters of d and d' to be identical.

Thus far, the subtyping relation is confined to datatypes. It may be extended to types in the usual (structural) way. In this paper, we force datatypes to be monotonic in their parameters. Hence, we can derive

$$\begin{array}{rcl} \texttt{Odd List} & \leq & \texttt{Nat List} \\ [\texttt{l1} : \texttt{Even}, \texttt{l2} : \texttt{Nat List}, \texttt{l3} : \texttt{Odd}] & \leq & [\texttt{l1} : \texttt{Nat}, \texttt{l2} : \texttt{Nat List}] \\ \texttt{Nat} \rightarrow \texttt{Even NeList} & \leq & \texttt{Odd} \rightarrow \texttt{Nat NeList} \end{array}$$

from the fact that `Odd` $\leq$ `Nat`, `Even` $\leq$ `Nat` and `'a NeList` $\leq$ `'a List`. The formal definition of the subtyping relation is presented in the next section.

In order to introduce *strict overloading*, which is a central concept in this paper, let us anticipate on the next section by considering the evaluation rule for case-expressions. Two observations can be made: first, our informal definition of datatype allows for arbitrary overloading of constructors. Second, it is not possible to define a type-independent evaluation rule for case-expressions for arbitrary datatypes. For example, consider the following datatype, where `Sum` is a datatype identifier of arity 2:

```
datatype ('a,'b) Sum = inj of 'a
                     | inj of 'b ;
```

Note that the datatype is obtained from the usual definition of sum types by overloading the constructors inj_1 and inj_2. Now, a case-expression for this datatype should be of the form

```
case a of (inj x) => b1 | (inj x) => b2
```

with evaluation rules

```
case (inj a) of (inj x) => b1 | (inj x) => b2  →  b1{x:=a}
case (inj a) of (inj x) => b1 | (inj x) => b2  →  b2{x:=a}
```

As `b1` and `b2` are arbitrary, the calculus is obviously not confluent. Thus one needs to impose some restrictions on overloading. One drastic solution to avoid non-confluence is to require constructors to be declared at most once in a given datatype, but this solution is too restrictive. A better solution is to require constructors to be declared "essentially" at most once in a given datatype. Here "essentially" consists in allowing a constructor `c` to be multiply defined in a datatype d, but by requiring that for every declaration `c of rho`, we have `rho` $\leq$ `rhom` where `c of rhom` is the first declaration of `c` in d. In other words, the only purpose of repeated declarations is to enforce the desired subtyping constraints but (once subtyping is defined) only the first declaration needs to be used for typing expressions. This notion, which we call strict overloading, is mild enough to be satisfied by most datatypes that occur in the literature, see [5] for a longer discussion on this issue.

We conclude this section with further examples of datatypes. Firstly, we define a datatype of ordinals (or better said of ordinal notations). Note that the datatype is a higher-order one, because of the constructor `lim` which takes a function as input.

```
datatype Ord = s of Ord | lim of (Nat -> Ord)
with     Nat ≤ Ord ;
```

Second, we define a datatype of binary integers. These datatypes are part of the Coq library, but Coq does not take advantage of constructor subtyping.

```
datatype positive = xH | xI of positive | xO of positive ;
datatype natural  = ZERO
with     positive ≤ natural ;
datatype integer  = NEG of positive
with     natural ≤ integer ;
```

Thirdly, and as pointed out in [5,12], constructor subtyping provides a suitable framework in which to formalize programming languages, including the object calculi of Abadi and Cardelli [1] and a variety of other languages taken from [29]. Yet another example of language that can be expressed with constructor semantics is mini-ML [22], as shown below. Here we consider four datatypes identifiers: `E` of *expressions*, `I` for *identifiers*, `P` of *patterns* and `N` for the *nullpattern*, all with arity 0.

```
datatype I = ident ;
datatype N = nullpat ;
datatype P = pairpat of (P * P)
with     I ≤ P, N ≤ P ;
datatype E = num | false | true | lamb of (P * E)
           | if of (E * E * E) | mlpair of (E * E)
           | apply of (E * E) | let of (P * E * E)
           | letrec of (P * E * E)
with     I ≤ E, N ≤ E ;
```

Lastly, we conclude with a definition of CTL* formulae, see [15]. In this example, we consider two datatypes identifiers `SF` of *state formulae* and `PF` of *path formulae*, both with arity 1.

```
datatype 'a SF = i of ('a * 'a SF) | conj of ('a SF * 'a SF)
               | not of 'a SF | forsomefuture of 'a PF
               | forallfuture of 'a PF
and      'a PF = conj of ('a PF * 'a PF) | not of 'a PF
               | nexttime of 'a PF | until of 'a PF
with     'a SF ≤ 'a PF ;
```

CTL* and related temporal logics provide suitable frameworks in which to verify the correctness of programs and protocols, and hence are interesting calculi to formalize in proof assistants.

3 A core calculus $\lambda_{\rightarrow,[],\mathsf{data}}$

In this section, we introduce the core calculus $\lambda_{\rightarrow,[],\mathsf{data}}$. The first subsection is devoted to types, datatypes and subtyping; the second subsection is devoted to expressions, reduction and typing.

3.1 Types and subtyping

Below we assume given some pairwise disjoint sets $\mathcal{L}$ of *labels*, $\mathcal{D}$ of *datatype identifiers*, $\mathcal{C}$ of *constructor identifiers* and $\mathcal{X}$ of *type variables*. Moreover, we let $l, l', l_i, \ldots$ range over $\mathcal{L}$, $d, d', \ldots$ range over $\mathcal{D}$, $c, c', c_i, \ldots$ range over $\mathcal{C}$ and $\alpha, \alpha', \alpha_i, \beta, \ldots$ range over $\mathcal{X}$. In addition, we assume that every datatype identifier d has a fixed *arity* $\mathsf{ar}(d)$ and that $\alpha_1, \alpha_2, \ldots$ is a fixed enumeration of $\mathcal{X}$.

Definition 1 (Types). *The set $\mathcal{T}$ of* types *is given by the abstract syntax:*

$$\sigma, \tau := d[\tau_1, \ldots, \tau_{\mathsf{ar}(d)}] \mid \alpha \mid \sigma \rightarrow \tau \mid [l_1 : \sigma_1, \ \ldots \ , l_n : \sigma_n]$$

where in the last clause it is assumed that the l_is are pairwise distinct. By convention, we identify record types that only differ in the order of their declarations, such as $[l : \sigma, l' : \tau]$ and $[l' : \tau, l : \sigma]$.

We now turn to the definition of datatype. Informally, a *datatype* is a list of *constructor declarations*, i.e. of pairs (c, τ) where c is a constructor identifier and τ is a *constructor type*, i.e. a type of the form

$$\rho_1 \rightarrow \ \ldots \ \rightarrow \rho_n \rightarrow d[\alpha_1, \ \ldots \ , \alpha_{\mathsf{ar}(d)}]$$

with $d \in \mathcal{D}$. However not all datatypes are valid. In order for a datatype to be valid, it must satisfy several properties.

1. Constructors must be *strictly positive*, so that datatypes have a direct set-theoretic interpretation. For example, $c_1 : \mathsf{nat} \rightarrow d$ and $c_2 : (\mathsf{nat} \rightarrow d) \rightarrow d$ are strictly positive w.r.t. d, whereas $c_3 : (d \rightarrow d) \rightarrow d$ is not.
2. Parameters must appear positively in the domains of constructor types, so that datatypes are *monotonic* in their parameters. For example, the parameter α appears positively in the domain of $\alpha \rightarrow d[\alpha]$, while it appears negatively in the domain of $(\alpha \rightarrow \mathsf{nat}) \rightarrow d[\alpha]$.
3. Datatypes that mutually depend on each other must have the same number of parameters, for the sake of simplicity.
4. Constructors must be strictly overloaded, so that case-expressions can be evaluated unambiguously.

In addition, we allow datatypes to depend on previously defined datatypes. This leads us naturally to the notion of *datatype context*. Informally, a datatype context is a finite list of datatypes. Below we let σ, τ range over types, $\aleph$ range over datatype contexts, c range over datatype constructors and d, d' range over datatype identifiers.

Definition 2.

1. *σ is a* legal type *in $\aleph$ with variables in $\{\alpha_1, \ \ldots \ , \alpha_k\}$ (or $\emptyset$ if $k = 0$), written $\aleph \vdash_k \sigma$ type, is defined by the rules of Figure 1;*
2. *σ is a* subtype *of τ in $\aleph$, written $\aleph \vdash \sigma \leq \tau$, is defined by the rules of Figure 2, where $\aleph \vdash d \leq d'$ if*
   - *$\mathsf{ar}(d) = \mathsf{ar}(d') = m$;*
   - *every declaration $c : \tau_1 \rightarrow \ \ldots \ \rightarrow \tau_n \rightarrow d[\alpha_1, \ \ldots \ , \alpha_m]$ in $\aleph$ is matched by another declaration $c : \tau_1 \rightarrow \ \ldots \ \rightarrow \tau_n \rightarrow d'[\alpha_1, \ \ldots \ , \alpha_m]$ in $\aleph$.*
3. *τ is a d-*constructor type *in $\aleph$, written $\aleph \vdash \tau \ \mathsf{coty}(d)$, is defined by the rules of Figure 3, where:*
   - *α appears positively in τ, written $\alpha \ \mathsf{pos} \ \tau$, is defined as in [17];*
   - *ρ is strictly positive w.r.t. d, written $\rho \ \mathsf{spos} \ d$, is defined as in [17];*
   - *$d \in \aleph$ if there exists a declaration $(c : \tau) \in \aleph$ in which d occurs;*
4. *$\aleph$ is a* legal datatype context, *written $\aleph \ \mathsf{legal}$, is defined by the rules of Figure 4, where $\aleph \ \mathsf{compatible} \ D, c : \tau$ if*
   - *for every $(c : \tau') \in D$, $\aleph \vdash \tau' \ \mathsf{coty}(d) \Rightarrow \aleph \vdash \tau' \leq \tau$;*
   - *for every $(c' : \tau') \in D$, $\aleph \vdash \tau' \ \mathsf{coty}(d') \Rightarrow \mathsf{ar}(d) = \mathsf{ar}(d')$.*

In addition, we say $c : \tau$ is a main d-declaration *if it is the first declaration of the form $c : \tau'$ with $\aleph \vdash \tau' \ \mathsf{coty}(d)$.*

A special case of constructor type is given by the rule

$$\frac{\aleph \vdash_0 \rho_i \ \mathsf{type} \ \vee \ \rho_i \in \{\alpha_1, \ldots, \alpha_{\mathsf{ar}(d)}, d[\alpha_1, \ \ldots \ , \alpha_k]\}}{\aleph \vdash \rho_1 \rightarrow \ \ldots \ \rightarrow \rho_n \rightarrow d[\alpha_1, \ \ldots \ , \alpha_k] \ \mathsf{coty}(d)} \quad d \notin \aleph$$

Note that conditions 3 and 4 above are enforced by the side-conditions in (add-cons) whereas conditions 1 and 2 above are enforced by the rule (coty). Also note that in the side condition for (add-cons), τ' and τ are compared w.r.t. $\aleph$ and not $\aleph; D$.

$$(\rightarrow) \qquad \frac{\aleph \vdash_k \sigma \ \mathsf{type} \qquad \aleph \vdash_k \tau \ \mathsf{type}}{\aleph \vdash_k \sigma \rightarrow \tau \ \mathsf{type}}$$

$$([]) \qquad \frac{\aleph \vdash_k \sigma_i \ \mathsf{type} \quad (1 \leq i \leq n)}{\aleph \vdash_k [l_1 : \sigma_1, \ \ldots \ , l_n : \sigma_n] \ \mathsf{type}}$$

$$(\mathsf{data}) \qquad \frac{d \in \aleph \quad \aleph \vdash_k \sigma_i \ \mathsf{type} \quad (1 \leq i \leq \mathsf{ar}(d))}{\aleph \vdash_k d[\sigma] \ \mathsf{type}}$$

$$(\alpha) \qquad \frac{\aleph \ \mathsf{legal}}{\aleph \vdash_k \alpha_i \ \mathsf{type}} \quad (1 \leq i < k)$$

Fig. 1. Type formation rules

$$(\leq_{\text{refl}}) \qquad \frac{\aleph \vdash_k \sigma \ \mathsf{type}}{\aleph \vdash \sigma \leq \sigma}$$

$$(\leq_{\text{trans}}) \qquad \frac{\aleph \vdash \sigma \leq \tau \quad \aleph \vdash \tau \leq \rho}{\aleph \vdash \sigma \leq \rho}$$

$$(\leq_{\to}) \qquad \frac{\aleph \vdash \sigma' \leq \sigma \quad \aleph \vdash \tau \leq \tau'}{\aleph \vdash \sigma \to \tau \leq \sigma' \to \tau'}$$

$$(\leq_{[]}) \qquad \frac{\aleph \vdash \sigma_i \leq \tau_i \quad (1 \leq i \leq n) \quad \aleph \vdash_k \sigma_j \ \mathsf{type} \quad (n+1 \leq j \leq m)}{\aleph \vdash [l_1 : \sigma_1, \ \ldots \ , l_{n+m} : \sigma_{n+m}] \leq [l_1 : \tau_1, \ \ldots \ , l_n : \tau_n]}$$

$$(\leq_{\mathsf{data}}) \qquad \frac{\aleph \vdash d \leq d' \quad \aleph \vdash \sigma_i \leq \tau_i \quad (1 \leq i \leq \mathsf{ar}(d))}{\aleph \vdash d[\boldsymbol{\sigma}] \leq d'[\boldsymbol{\tau}]}$$

Fig. 2. Subtyping rules

$$(\text{coty}) \qquad \frac{\aleph \vdash_k \rho_i \ \mathsf{type} \qquad \rho_i \ \mathsf{spos} \ d \qquad \alpha_j \ \mathsf{pos} \ \rho_i \quad (1 \leq i \leq n \,,\, 1 \leq j \leq k)}{\aleph \vdash \rho_1 \to \ \ldots \ \to \rho_n \to d[\alpha_1, \ \ldots \ , \alpha_k] \ \mathsf{coty}(d)} \quad d \notin \aleph$$

Fig. 3. Constructor type rule

3.2 Expressions and typing

In this subsection, we conclude the definition of $\lambda_{\to,[],\mathsf{data}}$ by defining its expressions, specifying their computational behavior and providing them with a typing system. Below we assume given a set $\mathcal{V}$ of *variables* and let $x, x', x_i, y, \ldots$ range over $\mathcal{V}$. Moreover, we assume given a legal datatype context $\aleph$ and let $\mathcal{T}_0$ be the set of legal types in $\aleph$; finally $\sigma, \tau, \ldots$ are assumed to range over $\mathcal{T}_0$.

Definition 3. *The set $\mathcal{E}$ of* expressions *is given by the abstract syntax:*

$$a, b := x \mid \lambda x{:}\tau.\, a \mid a\ b \mid [l_1 = a_1, \ \ldots \ , l_n = a_n] \mid a.l \mid$$
$$c[\boldsymbol{\sigma}]\ \boldsymbol{a} \mid \mathsf{case}^{\tau}_{d[\boldsymbol{\sigma}]}\ a\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ \ldots \ \mid c_n \Rightarrow b_n\}$$

$$(\text{empty}) \qquad . \ \mathsf{legal}$$

$$(\text{close}) \qquad \frac{\aleph; D \ \mathsf{legal}}{\aleph; D; \ \mathsf{legal}}$$

$$(\text{add-cons}) \qquad \frac{\aleph; D \ \mathsf{legal} \quad \aleph \vdash \tau \ \mathsf{coty}(d)}{\aleph; D, c : \tau \ \mathsf{legal}} \quad \aleph \ \mathsf{compatible} \ D, c : \tau$$

Fig. 4. Datatype rules

Free and bound variables, substitution $.\{. := .\}$ are defined the usual way. Moreover we assume standard variable conventions [2] and identify record expressions which only differ in the order of their components, e.g. $[l = a, l' = a']$ and $[l' = a', l = a]$. All the constructions are the usual ones, except perhaps for case-expressions, which are typed so as to avoid failure of subject reduction, see e.g. [19], and are slightly different from the usual case expressions in that we pattern-match against constructors rather than against patterns.

Definition 4 (Typing).

1. *A* context Γ *is a finite set of assumptions* $x_1 : \tau_1, \ \ldots, x_n : \tau_n$ *such that the* x_i*s are pairwise distinct elements of* $\mathcal{V}$ *and* $\tau_i \in \mathcal{T}_0$.
2. *A* judgment *is a triple of the form* $\Gamma \vdash a : \tau$, *where* Γ *is a context,* $a \in \mathcal{E}$ *and* $\tau \in \mathcal{T}_0$.
3. *A judgment is* derivable *if it can be inferred from the rules of Figure 5, where in the (case) rule it is assumed that* $c_1 : \tau_1, \ \ldots, c_n : \tau_n$ *are the sole main d-declarations and that* τ^σ *denotes* $\xi_1 \to \ \ldots \ \to \xi_n \to \sigma$ *whenever* $\tau = \xi_1 \to \ \ldots \ \to \xi_n \to d[\boldsymbol{\rho}]$.
4. *An expression* $a \in \mathcal{E}$ *is* typable *if* $\Gamma \vdash a : \sigma$ *for some context* Γ *and type* σ.

(start) $$\Gamma \vdash x : \tau \qquad \text{if } x : \tau \in \Gamma$$

(application) $$\frac{\Gamma \vdash e : \tau \to \sigma \quad \Gamma \vdash e' : \tau}{\Gamma \vdash e\, e' : \sigma}$$

(abstraction) $$\frac{\Gamma, x : \tau \vdash e : \sigma}{\Gamma \vdash \lambda x{:}\tau.\, e : \tau \to \sigma}$$

(record) $$\frac{\Gamma \vdash e_i : \tau_i \quad (1 \le i \le n)}{\Gamma \vdash [l_1 = e_1, \ \ldots, l_n = e_n] : [l_1 : \tau_1, \ \ldots, l_n : \tau_n]}$$

(select) $$\frac{\Gamma \vdash e : [l_1 : \tau_1, \ \ldots, l_n : \tau_n]}{\Gamma \vdash e.l_i : \tau_i} \qquad \text{if } 1 \le i \le n$$

(constructor) $$\frac{\Gamma \vdash b_i : \rho_i\{\boldsymbol{\alpha} := \boldsymbol{\tau}\} \quad (1 \le i \le k)}{\Gamma \vdash c[\boldsymbol{\tau}]\ \boldsymbol{b} : d[\boldsymbol{\tau}]} \quad \text{if } c : \rho_1 \to \ldots \to \rho_k \to d[\boldsymbol{\alpha}] \in \aleph$$

(case) $$\frac{\Gamma \vdash a : d[\boldsymbol{\rho}] \quad \Gamma \vdash b_i : (\tau_i\{\boldsymbol{\alpha} := \boldsymbol{\rho}\})^\sigma \quad (1 \le i \le n)}{\Gamma \vdash \mathsf{case}^\sigma_{d[\boldsymbol{\rho}]}\ a\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ \ldots \ \mid c_n \Rightarrow b_n\} : \sigma}$$

(subsumption) $$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e : \sigma} \qquad \text{if } \tau \le \sigma$$

Fig. 5. Typing rules

The computational behavior of $\lambda_{\rightarrow,[],\mathsf{data}}$ is drawn from the usual notion of β-reduction, ι-reduction and π-reduction.

Definition 5.

1. *β-reduction $\rightarrow_\beta$ is defined as the compatible closure of the rule*

$$(\lambda x{:}\sigma.\, a)\, b \rightarrow_\beta a\{x := b\}$$

2. *π-reduction $\rightarrow_\pi$ is defined as the compatible closure of the rule*

$$[l_1 = a_1,\ \ldots\ , l_n = a_n].l_i \rightarrow_\pi a_i$$

3. *ι-reduction $\rightarrow_\iota$ is defined as the compatible closure of the rule*

$$\mathsf{case}^{\sigma}_{d[\boldsymbol{\tau}']}\ (c_i[\boldsymbol{\tau}]\ \boldsymbol{a})\ \mathsf{of}\ \{c_1 \Rightarrow f_1 \mid\ \ldots\ \mid c_n \Rightarrow f_n\} \rightarrow_\iota f_i\ \boldsymbol{a}$$

4. *$\rightarrow_{basic}$ is defined as $\rightarrow_\beta \cup \rightarrow_\pi \cup \rightarrow_\iota$.*
5. *$\twoheadrightarrow_{basic}$ and $=_{basic}$ are respectively defined as the reflexive-transitive and the reflexive-symmetric-transitive closures of $\rightarrow_{basic}$.*

Note that we do not require $\boldsymbol{\tau}$ and $\boldsymbol{\tau'}$ to coincide in the definition of ι-reduction as it would lead to too weak an equational theory. However, the typing rules will enforce $\boldsymbol{\tau} \leq \boldsymbol{\tau'}$ on legal terms.

4 Meta-theory of the core language

In this section, we summarize some basic properties of the core language.

Proposition 1 (Confluence). *$\rightarrow_{basic}$ is confluent:*

$$a =_{basic} b \quad \Rightarrow \quad \exists c \in \mathcal{E}.\ a \twoheadrightarrow_{basic} c \quad \wedge \quad b \twoheadrightarrow_{basic} c$$

Proof. By the standard technique of Tait and Martin-Löf.

Proposition 2 (Subject reduction). *Typing is closed under $\rightarrow_{basic}$:*

$$\Gamma \vdash a : \sigma \quad \wedge \quad a \rightarrow_{basic} b \quad \Rightarrow \quad \Gamma \vdash b : \sigma$$

Proof. By induction on the structure of the derivations, using some basic properties of subtyping.

As usual, we say that an expression e is *strongly normalizing* with respect to a relation $\rightarrow$ if there is no infinite sequence

$$e \rightarrow e_1 \rightarrow e_2 \rightarrow\ \ldots$$

We let $\mathsf{SN}(\rightarrow)$ denote the set of expressions that are strongly normalizing with respect to $\rightarrow$.

Proposition 3 (Strong normalization). *$\rightarrow_{basic}$ is strongly normalizing on typable expressions:*

$$\Gamma \vdash a : \sigma \quad \Rightarrow \quad a \in \mathsf{SN}(\rightarrow_{basic})$$

Proof. By a standard computability argument.

We now turn to type-checking. One cannot rely on the existence of minimal types, as they may not exist (for minimal types to exist, one must require datatypes to be pre-regular, see e.g. [5,18]). Instead, we can define for every context Γ and expression a a finite set $\mathsf{min}_\Gamma(a)$ of minimal types such that

$$\begin{array}{rcl} \sigma \in \mathsf{min}_\Gamma(a) & \Rightarrow & \Gamma \vdash a : \sigma \\ \Gamma \vdash a : \sigma & \Rightarrow & \exists \tau \in \mathsf{min}_\Gamma(a).\ \tau \leq \sigma \end{array}$$

The set $\mathsf{min}_\Gamma(a)$, which is defined in the obvious way, is finite because there are only finitely many declarations for each constructor.

Proposition 4. *Type-checking is decidable: there exists an algorithm to decide whether a given judgment $\Gamma \vdash a : \sigma$ is derivable.*

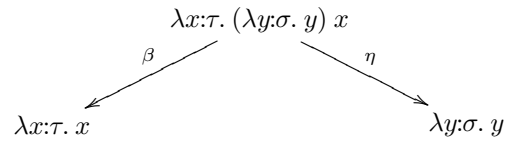
Proof. Proceed in two steps: first compute $\mathsf{min}_\Gamma(a)$, second check whether there exists $\tau \in \mathsf{min}_\Gamma(a)$ such that $\tau \leq \sigma$.

5 Extensionality

5.1 Motivations

Extensionality, as embodied e.g. in η-conversion, is a basic feature of many type systems. Traditionally, extensionality equalities are oriented as contractive rules: e.g. η-conversion is oriented as η-reduction. On the other hand, expansive rules provide an alternative computational interpretation of extensionality equalities: e.g. η-conversion may be oriented as η-expansion. Expansive extensionality rules have numerous applications in categorical rewriting, unification and partial evaluation. In addition to these traditional motivations, which are nicely summarized in [13], subtyping adds some new fundamental reasons to use expansive rules:

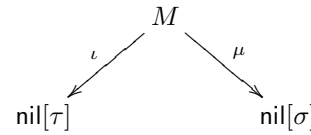
1. contractive rules lead to non-confluent calculi, even on well-typed expressions: if we adopt η-reduction for λ-abstractions, then the following critical pair cannot be solved:



On the other hand, $\lambda x{:}\tau.\ (\lambda y{:}\sigma.\ y)\ x$ is well-typed (of type $\tau \to \sigma$) whenever $\tau \leq \sigma$ (this observation is due to Mitchell, Hoang and Howard [27]). A similar remark applies to datatypes: if we adopt μ-reduction for lists, as defined by

$$\mathsf{case}^{\mathsf{list}[\tau]}_{\mathsf{list}[\tau]}\ e\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow \mathsf{nil}[\tau] \mid \mathsf{cons} \Rightarrow \lambda a{:}\tau.\ \lambda l{:}\mathsf{list}[\tau].\ \mathsf{cons}[\tau] \mathsf{a}\ \mathsf{l}\} \to_\mu \mathsf{e}$$

then the following critical pair cannot be solved:



where $M \equiv \mathsf{case}^{\mathsf{list}[\tau]}_{\mathsf{list}[\tau]}\ (\mathsf{nil}[\sigma])\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow \mathsf{nil}[\tau] \mid \mathsf{cons} \Rightarrow \lambda a{:}\tau.\ \lambda l{:}\mathsf{list}[\tau].\ \mathsf{cons}[\tau] \mathsf{a}\ \mathsf{l}\}$. On the other hand, $\mathsf{case}^{\mathsf{list}[\tau]}_{\mathsf{list}[\tau]}\ (\mathsf{nil}[\sigma])\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow \mathsf{nil}[\tau] \mid \mathsf{cons} \Rightarrow \lambda a{:}\tau.\ \lambda l{:}\mathsf{list}[\tau].\ \mathsf{cons}[\tau] \mathsf{a}\ \mathsf{l}\}$ is well-typed (of type $\mathsf{list}[\tau]$) whenever $\sigma \leq \tau$.

2. contractive rules lead to calculi with too many canonical inhabitants (i.e. closed expressions in normal form): if we adopt μ-reduction for lists then the following expressions are canonical inhabitants of $\mathsf{list}[\tau]$, provided $\sigma \leq \tau$, $a : \sigma$ and $l : \mathsf{list}[\sigma]$:

$$\mathsf{nil}[\sigma] \qquad \mathsf{nil}[\tau] \qquad \mathsf{cons}[\sigma] a\ l \qquad \mathsf{cons}[\tau] a\ l$$

On the other hand, one would expect canonical inhabitants of $\mathsf{list}[\tau]$ to be of the form

$$\mathsf{nil}[\tau] \qquad \mathsf{cons}[\tau] a\ l$$

where in the second case l itself is a canonical inhabitant of $\mathsf{list}[\tau]$ and a is a canonical inhabitant of τ. Remarkably we obtain the desired effect if we reverse μ-reduction. With this new reduction rule, which we call μ-expansion and denote by $\to_{\overline{\mu}}$, we have:

$$\begin{aligned} \mathsf{nil}[\sigma] &\to_{\overline{\mu}} \mathsf{case}^{\mathsf{list}[\tau]}_{\mathsf{list}[\tau]}\ \mathsf{nil}[\sigma]\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow \mathsf{nil}[\tau] \mid \mathsf{cons} \Rightarrow \mathsf{cons}[\tau]\} \\ &\to_\iota \mathsf{nil}[\tau] \end{aligned}$$

Similarly, for $a : \sigma$ and $l : \mathsf{list}[\sigma]$, one has:

$$\begin{aligned} \mathsf{cons}[\sigma] a\ l &\to_{\overline{\mu}} \mathsf{case}^{\mathsf{list}[\tau]}_{\mathsf{list}[\tau]}\ (\mathsf{cons}[\sigma] a\ l)\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow \mathsf{nil}[\tau] \mid \mathsf{cons} \Rightarrow \mathsf{cons}[\tau]\} \\ &\to_\iota \mathsf{cons}[\tau]\ a\ l \end{aligned}$$

(Strictly speaking, expansive extensionality rules are defined relative to a context and a type and the above reductions are performed at type $\mathsf{list}[\tau]$);

3. expansive rules provide a simple but useful program optimization: if we adopt expansive rules for records, the expression $[n = 3, c = \mathsf{blue}]$ reduces at type $[n : \mathsf{nat}]$ to $[n = 3]$, thus throwing out the irrelevant fields at type $[n : \mathsf{nat}]$.

We therefore embark upon studying an expansive interpretation of extensionality in $\lambda_{\to,[],\mathsf{data}}$.

5.2 Expansive extensionality rules

The computational behavior of the calculus is now obtained by aggregating the expansive extensionality rules to $\rightarrow_{basic}$. Expansive extensionality rules need to be formulated in a typed framework so we consider judgments of the form

$$\Gamma \vdash a \rightarrow b : \sigma$$

For the sake of uniformity, we first reformulate $\rightarrow_{basic}$ in a typed framework.

Definition 6.

1. Typed *basic*-reduction $\rightarrow_{basic}$ *is defined by the clause*

$$\Gamma \vdash a \rightarrow_{basic} b : \sigma$$

iff $\Gamma \vdash a : \sigma$ *and* $a \rightarrow_{basic} b$.

2. η-expansion $\rightarrow_{\overline{\eta}}$ *is defined as the quasi-compatible closure (see below) of the rule*

$$\Gamma \vdash a \rightarrow_{\overline{\eta}} \lambda x{:}\tau.\, a\, x : \tau \rightarrow \sigma$$

provided $a \neq \lambda x{:}\tau.\, b$. *The usual rule*

$$\frac{\Gamma \vdash a \rightarrow_{\overline{\eta}} b : \tau \rightarrow \sigma \qquad \Gamma \vdash c : \tau}{\Gamma \vdash a\ c \rightarrow_{\overline{\eta}} b\ c : \sigma}$$

is only allowed under the proviso $b \neq \lambda x{:}\tau.\, a\, x$.

3. Surjective pairing $\rightarrow_{sp}$ *is defined as the quasi-compatible closure (see below) of the rule*

$$\Gamma \vdash a \rightarrow_{sp} [l_1 = a.l_1,\ \ldots\ , l_n = a.l_n] : [l_1 : \tau_1,\ \ldots\ , l_n : \tau_n]$$

provided $a \neq [l_1 = a_1,\ \ldots\ , l_n = a_n]$. *The usual rule*

$$\frac{\Gamma \vdash a \rightarrow_{sp} b : [l_1 : \tau_1,\ \ldots\ , l_n : \tau_n]}{\Gamma \vdash a.l_i \rightarrow_{sp} b.l_i : \tau_i}$$

is only allowed under the proviso $b \neq [l_1 = a.l_1,\ \ldots\ , l_n = a.l_n]$.

4. μ-expansion $\rightarrow_{\overline{\mu}}$ *is defined as the quasi-compatible closure (see below) of the rule*

$$\Gamma \vdash a \rightarrow_{\overline{\mu}} \mathsf{case}^{d[\boldsymbol{\tau}]}_{d[\boldsymbol{\tau}]}\ a\ \mathsf{of}\ \{c_1 \Rightarrow c_1[\boldsymbol{\tau}] \mid \ \ldots\ \mid c_n \Rightarrow c_n[\boldsymbol{\tau}]\} : d[\boldsymbol{\tau}]$$

provided $a \neq c_i[\boldsymbol{\tau}]\boldsymbol{b}$ *and* $a \neq \mathsf{case}^{d[\boldsymbol{\tau}]}_{d[\boldsymbol{\tau}]}\ a'\ \mathsf{of}\ \{c_1 \Rightarrow c_1[\boldsymbol{\tau}] \mid \ \ldots\ \mid c_n \Rightarrow c_n[\boldsymbol{\tau}]\}$. *The usual rule*

$$\frac{\Gamma \vdash a \rightarrow_{\overline{\mu}} a' : d[\boldsymbol{\tau}]}{\Gamma \vdash \mathsf{case}^{\sigma}_{d[\boldsymbol{\tau}]}\ a\ \mathsf{of}\ \{\boldsymbol{c} \Rightarrow \boldsymbol{b}\} \rightarrow_{\overline{\mu}} \mathsf{case}^{\sigma}_{d[\boldsymbol{\tau}]}\ a'\ \mathsf{of}\ \{\boldsymbol{c} \Rightarrow \boldsymbol{b}\} : \sigma}$$

is only allowed under the proviso $a' \neq \mathsf{case}^{d[\boldsymbol{\tau}]}_{d[\boldsymbol{\tau}]}\ a\ \mathsf{of}\ \{c_1 \Rightarrow c_1[\boldsymbol{\tau}] \mid \ \ldots\ \mid c_n \Rightarrow c_n[\boldsymbol{\tau}]\}$.

5. Typed $full$-reduction $\rightarrow_{full}$ *is defined as the union of basic,* $\overline{\eta}$, sp, $\overline{\mu}$*-reduction, i.e.*
$$\Gamma \vdash a \rightarrow_{full} b : \sigma \quad \Leftrightarrow \quad \Gamma \vdash a \rightarrow_{basic,\overline{\eta},sp,\overline{\mu}} b : \sigma$$
6. $\twoheadrightarrow_{full}$ *and* $=_{full}$ *are respectively defined as the reflexive-transitive and the reflexive-symmetric-transitive closures of* $\rightarrow_{full}$.

Several points deserve attention:

1. the various restrictions on $\rightarrow_{\overline{\eta}}$, $\rightarrow_{sp}$ and $\rightarrow_{\overline{\mu}}$ are required to enforce strong normalization. Without those restrictions, one would have loops or infinite reductions, see the appendix.
2. unlike the traditional formulations of η-expansion, we do allow η-expansions on λ-abstractions at type $\tau \rightarrow \sigma$ if the type of the variable is not τ. Such a possibility is indeed crucial for expressions of type $\sigma \rightarrow \tau$ to reduce to an expression of the form $\lambda x\!:\!\sigma.\ e$ at that type. On the other hand, note that η-expansion as defined here does not preserve $\rightarrow_{basic}$-normal forms. For example, for $\tau \leq \sigma$,
$$\vdash \lambda x{:}\sigma.\ x : \tau \rightarrow \sigma$$
is in $\rightarrow_{basic}$-normal form but
$$\begin{aligned} \vdash \lambda x{:}\sigma.\ x &\rightarrow_{\overline{\eta}} \lambda z{:}\tau.\ (\lambda x{:}\sigma.\ x)\ z\ : \tau \rightarrow \sigma \\ &\rightarrow_{\beta} \lambda z{:}\tau.\ z \end{aligned}$$
A similar remark applies to records and case-expressions.
3. $\rightarrow_{\overline{\mu}}$-like rules for datatypes seem to have received very little attention in the literature. As far as we know, only Ghani [16] proposes a possible such rule (his rule is motivated by categorical considerations) but does not study it in detail. Our expansion rule for datatypes is weaker than the one suggested by Ghani [16] and thus is inadequate to capture the categorical view of datatypes as initial algebras in a suitable category. It nevertheless serves its purpose, see Proposition 7.
4. reduction is not preserved under subsumption: that is, one may have
$$\Gamma \vdash a \rightarrow_{full} b : \sigma \quad \wedge \quad \Gamma \not\vdash a \rightarrow_{full} b : \tau$$
for $\sigma \leq \tau$. On the other hand,
$$\Gamma \vdash a \rightarrow_{full} b : \sigma \quad \Rightarrow \quad \Gamma \vdash a =_{full} b : \tau$$
for $\sigma \leq \tau$.

5.3 Preservation of confluence and strong normalization

Expansive extensionality rules preserve the fundamental properties of $\lambda_{\rightarrow,[],\mathsf{data}}$.

Proposition 5 (Strong normalization). *The relation* $\rightarrow_{full}$ *is strongly normalizing on typable expressions.*

Proof. By modifying, along the lines of e.g. [20], the computability argument of Theorem 3.

Proposition 6 (Confluence). *The relation $\rightarrow_{full}$ is confluent on typable expressions.*

Proof. Using Newman's Lemma, strong normalization and weak confluence, which is proved by a case analysis on the possible critical pairs.

5.4 Theory of canonical inhabitants

Below we write $\Gamma \vdash^{nf} a : \tau$ if $\Gamma \vdash a : \tau$ and there is no $b \in \mathcal{E}$ such that $\Gamma \vdash a \rightarrow_{full} b : \tau$. The following result shows that the theory of canonical inhabitants is well-behaved, i.e. that typable closed expressions in normal form have the expected shape.

Proposition 7. *Assume that $\Gamma \vdash^{nf} a : \tau$.*

1. *If $\tau = \sigma \rightarrow \rho$, then $a = \lambda x{:}\sigma.\, b$;*
2. *If $\tau = [l_1 : \sigma_1, \ \ldots \ , l_n : \sigma_n]$, then $a = [l_1 = b_1, \ \ldots \ , l_n = b_n]$.*
3. *If $\tau = d[\boldsymbol{\sigma}]$, then $a = c[\boldsymbol{\sigma}]\boldsymbol{b}$.*

Proof. By a case analysis on the possible normal forms.

The above result may be seen as evidence that the $\overline{\eta}, sp, \overline{\mu}$-rules restore a semantical justification of the system, and in particular of the case-expressions: as every canonical inhabitant of $d[\boldsymbol{\tau}]$ is of the form $c[\boldsymbol{\tau}]\boldsymbol{b}$, it is justified to do pattern-matching on c.

6 Adding fixpoints

$\lambda_{\rightarrow,[],\mathsf{data}}$ has a very restricted computational power. In particular, it does not support recursion. In this section, we study an extension of $\lambda_{\rightarrow,[],\mathsf{data}}$ with fixpoints, and show the resulting calculus to be confluent.

Definition 7.

1. *The set of expressions $\mathcal{E}$ is extended with the clause* $\mathsf{fix}\ x{:}\tau.\, a$.
2. *Fixpoint reduction $\rightarrow_{rec}$ is defined as the compatible closure of the rule*

$$\mathsf{fix}\ x{:}\tau.\, a \rightarrow_{rec} a\{x := \mathsf{fix}\ x{:}\tau.\, a\}$$

3. *The typing system is extended with the rule:*

$$\frac{\Gamma, x : \tau \vdash a : \tau}{\Gamma \vdash \mathsf{fix}\ x{:}\tau.\, a : \tau}$$

4. *We let $\rightarrow_{full+rec}$ denote $\rightarrow_{full} \cup \rightarrow_{rec}$.*

We have:

Proposition 8. *The relation $\rightarrow_{full+rec}$ is confluent on typable expressions.*

Proof. Using a standard technique due to Lévy [23], and exploited e.g. in [14]. The idea is to introduce bounded fixpoints, show that the calculus remains strongly normalizing and confluent, and then use some elementary reasoning on abstract reduction systems to conclude that $\rightarrow_{full+rec}$ is confluent.

Obviously, $\rightarrow_{full+rec}$ is not strongly normalizing. In order to preserve strong normalization, one must restrict oneself to *guarded* fix-expressions. Technically, it is achieved by defining the notion of an expression e being guarded, and by adding the side-condition a is guarded in the typing rule for fixpoints. A precise description of the guard mechanism may be found for example in [17].

7 Conclusion and directions for further work

In this paper, we have introduced a simply typed λ-calculus with record types and parametric datatypes. The calculus supports a combination of record subtyping and constructor subtyping and thus provides a flexible type system. We have shown the calculus to be well-behaved, in particular with respect to canonical inhabitants.

In the future, we intend to study *definitions* for $\lambda_{\rightarrow,[],\mathsf{data}}$ and its extensions. Our goal is to aggregate a theory of definitions which is flexible enough to support overloaded definitions, such as multiplication $*$:

$$\begin{aligned} * &= *_1 : \mathbb{N} \rightarrow \mathbb{E} \rightarrow \mathbb{E} \\ &= *_2 : \mathbb{E} \rightarrow \mathbb{N} \rightarrow \mathbb{E} \\ &= *_3 : \mathbb{O} \rightarrow \mathbb{O} \rightarrow \mathbb{O} \\ &= *_4 : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \end{aligned}$$

where each $*_i$ is defined using case-expressions and recursion. As suggested by the above example, the idea is to allow identifiers to stand for several functions that have a different type. To do so, several options exist: for example, one may require the definitions to be coherent in a certain sense. Alternately, one may exploit some strategy, see e.g. [10,21], to disambiguate the definitions. Both approaches deserve further study.

Furthermore, we intend to scale up the results of this paper to more complex type systems.

1. Type systems for programming languages: in line with recent work on the design of higher-order typed (HOT) languages, one may envisage extending $\lambda_{\rightarrow,[],\mathsf{data}}$ with further constructs, including bounded quantification [9], objects [1], bounded operator abstraction [11]. We are also interested in scaling up our results to programming languages with dependent types such as DML [33]. The DML type system is based on constraints, and hence it seems possible to consider constructor subtyping on inductive families, as for example

in $\mathsf{X}\ i \leq \mathsf{X}\ j$ if $i \leq j$ where $\mathsf{X}\ i$ is the type $\{0, \ldots, i\}$. Extending constructor subtyping to inductive families is particularly interesting to implement type systems with subtyping.
2. Type systems for proof assistants: the addition of subtyping to proof assistants has been a major motivation for this work. Our next step is to investigate an extension of the Calculus of Inductive/Coinductive Constructions, see e.g. [17], with constructor subtyping. As suggested in [5,12], such a calculus seems particularly appropriate to formalize Kahn's natural semantics [22].

In yet a different direction, it may be interesting to study *destructor subtyping*, a dual to constructor subtyping, in which an inductive type σ is a subtype of another inductive type τ if σ has more destructors than τ. The primary example of destructor subtyping is of course record subtyping, as found in this paper. We leave for future work the study of destructor subtyping and of its interaction with constructor subtyping.

References

1. M. Abadi and L. Cardelli. *A theory of objects*. Springer-Verlag, 1996.
2. H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, revised edition, 1984.
3. H. Barendregt. The impact of the lambda calculus in logic and computer science. *Bulletin of Symbolic Logic*, 3(2):181–215, June 1997.
4. B. Barras *et. al. The Coq Proof Assistant User's Guide. Version 6.2*, May 1998.
5. G. Barthe. Order-sorted inductive types. *Information and Computation*, 199x. To appear.
6. G. Barthe and M.J. Frade. Constructor subtyping. Technical Report UMDITR9807, Department of Computer Science, University of Minho, 1998.
7. G. Betarte. *Dependent Record Types and Algebraic Structures in Type Theory*. PhD thesis, Department of Computer Science, Chalmers Tekniska Högskola, 1998.
8. L. Cardelli. Type systems. *ACM Computing Surveys*, 28(1):263–264, March 1996.
9. L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.
10. G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, February 1995.
11. A. Compagnoni and H. Goguen. Typed operational semantics for higher order subtyping. Technical Report ECS-LFCS-97-361, University of Edinburgh, July 1997.
12. T. Coquand. Pattern matching with dependent types. In B. Nordström, editor, *Informal proceedings of Logical Frameworks'92*, pages 66–79, 1992.
13. R. Di Cosmo. A brief history of rewriting with extensionality. Presented at the International Summer School on Type Theory and Term Rewriting, Glasgow, September 1996.
14. R. Di Cosmo and D. Kesner. Simulating expansions without expansions. *Mathematical Structures in Computer Science*, 4(3):315–362, September 1994.
15. E. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of theoretical computer science*, volume B, pages 995–1072. Elsevier Publishing, 1990.

16. N. Ghani. *Adjoint rewriting*. PhD thesis, Laboratory for the Foundations of Computer Science, University of Edinburgh, 1995.
17. E. Giménez. Structural recursive definitions in Type Theory. In K.G. Larsen, S. Skyum, and G. Winskel, editors, *Proceedings of ICALP'98*, volume 1443 of *Lecture Notes in Computer Science*, pages 397–408. Springer-Verlag, 1998.
18. J. Goguen and R. Diaconescu. An Oxford survey of order sorted algebra. *Mathematical Structures in Computer Science*, 4(3):363–392, September 1994.
19. H. Hosoya, B. Pierce, and D.N. Turner. Subject reduction fails in Java. Message to the TYPES mailing list, 1998.
20. C.B. Jay and N. Ghani. The virtues of eta-expansion. *Journal of Functional Programming*, 5(2):135–154, April 1995.
21. M.P. Jones. Dictionary-free overloading by partial evaluation. In *Proceedings of PEPM'94*, pages 107–117, 1994. University of Melbourne, Australia, Department of Computer Science, Technical Report 94/9.
22. G. Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 1987.
23. J.-J. Lévy. An algebraic interpretation of the $\lambda\beta\kappa$-calculus and a labelled λ-calculus. *Theoretical Computer Science*, 2:97–114, 1976.
24. Z. Luo. Coercive subtyping. *Journal of Logic and Computation*, 199x. To appear.
25. S. Marlow and P. Wadler. A practical subtyping system for Erlang. In *Proceedings of ICFP'97*, pages 136–149. ACM Press, 1997.
26. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.
27. J. C. Mitchell, M. Hoang, and B. T. Howard. Labelling techniques and typed fixed-point operators. In A.D. Gordon and A.M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, pages 137–174. Cambridge University Press, 1998.
28. J. Peterson and K. Hammond (editors). *Haskell 1.4.: A Non-strict, Purely Functional Language*, April 1997.
29. F. Pfenning. Refinement types for logical frameworks. In H. Geuvers, editor, *Informal Proceedings of TYPES'93*, pages 285–299, 1993.
30. B.C. Pierce and D.N. Turner. Local type inference. In *Proceedings of POPL'98*, pages 252–265. ACM Press, 1998.
31. F. Pottier. *Synthèse de types en présence de sous-typage: de la théorie la pratique*. PhD thesis, Université Paris VII, 1998.
32. N. Shankar, S. Owre, and J.M. Rushby. *The PVS Proof Checker: A Reference Manual*. Computer Science Laboratory, SRI International, February 1993. Supplemented with the PVS2 Quick Reference Manual, 1997.
33. H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of POPL'99*. ACM Press, 1999. To appear.

Loops and infinite reductions for unrestricted extensionality rules

For η-expansion:

$$\begin{aligned} \Gamma \;\vdash\; a\ c &\to_{\overline{\eta}} (\lambda x{:}\tau.\, a\ x)\ c : \tau \to \sigma \\ &\to_{\beta} a\ c \end{aligned}$$

For surjective pairing (we treat the case where $a : [l : \tau, l' : \sigma]$ but a similar remark applies to arbitrary records):

$$\begin{aligned} \Gamma \vdash\ & a.l \to_{sp} [l = a.l, l' = a.l'].l : \tau \\ & \to_{\pi}\ a.l \end{aligned}$$

For μ-expansion (if we allow constructors to be expanded):

$$\begin{aligned} \Gamma \vdash\ & (c_i[\boldsymbol{\tau}]\ \boldsymbol{b}) \to_{\overline{\mu}} \mathsf{case}_{d[\boldsymbol{\tau}]}^{d[\boldsymbol{\tau}]}\ (c_i[\boldsymbol{\tau}]\ \boldsymbol{b})\ \mathsf{of}\ \{c_1 \Rightarrow c_1[\boldsymbol{\tau}] \mid \ \ldots\ \mid c_n \Rightarrow c_n[\boldsymbol{\tau}]\} : d[\boldsymbol{\tau}] \\ & \to_{\iota}\ c_i[\boldsymbol{\tau}]\ \boldsymbol{b} \end{aligned}$$

and (if we allow case-expressions to be expanded):

$$\begin{aligned} \Gamma \vdash \mathsf{case}_{d[\boldsymbol{\tau}]}^{d[\boldsymbol{\tau}]}\ a\ \mathsf{of}\ \{c_1 \Rightarrow c_1[\boldsymbol{\tau}] \mid \ \ldots\ \mid c_n \Rightarrow c_n[\boldsymbol{\tau}]\} & : d[\boldsymbol{\tau}] \\ \to_{\overline{\mu}}\ \ \mathsf{case}_{d[\boldsymbol{\tau}]}^{d[\boldsymbol{\tau}]}\ a_1\ \mathsf{of}\ \{c_1 \Rightarrow c_1[\boldsymbol{\tau}] \mid \ \ldots\ \mid c_n \Rightarrow c_n[\boldsymbol{\tau}]\} & \\ \to_{\overline{\mu}}\ \ \mathsf{case}_{d[\boldsymbol{\tau}]}^{d[\boldsymbol{\tau}]}\ a_2\ \mathsf{of}\ \{c_1 \Rightarrow c_1[\boldsymbol{\tau}] \mid \ \ldots\ \mid c_n \Rightarrow c_n[\boldsymbol{\tau}]\} & \\ \to_{\overline{\mu}}\ \ \ldots & \end{aligned}$$

where $a_0 = a$ and

$$a_{i+1} = \mathsf{case}_{d[\boldsymbol{\tau}]}^{d[\boldsymbol{\tau}]}\ a_i\ \mathsf{of}\ \{c_1 \Rightarrow c_1[\boldsymbol{\tau}] \mid \ \ldots\ \mid c_n \Rightarrow c_n[\boldsymbol{\tau}]\}$$

and (if we take the compatible closure of $\overline{\mu}$):

$$\begin{aligned} \Gamma \vdash\ & a \to_{\overline{\mu}} \mathsf{case}_{d[\boldsymbol{\tau}]}^{d[\boldsymbol{\tau}]}\ a\ \mathsf{of}\ \{c_1 \Rightarrow c_1[\boldsymbol{\tau}] \mid \ \ldots\ \mid c_n \Rightarrow c_n[\boldsymbol{\tau}]\} \ : d[\boldsymbol{\tau}] \\ & \to_{\overline{\mu}} \mathsf{case}_{d[\boldsymbol{\tau}]}^{d[\boldsymbol{\tau}]}\ a_1\ \mathsf{of}\ \{c_1 \Rightarrow c_1[\boldsymbol{\tau}] \mid \ \ldots\ \mid c_n \Rightarrow c_n[\boldsymbol{\tau}]\} \\ & \to_{\overline{\mu}} \mathsf{case}_{d[\boldsymbol{\tau}]}^{d[\boldsymbol{\tau}]}\ a_2\ \mathsf{of}\ \{c_1 \Rightarrow c_1[\boldsymbol{\tau}] \mid \ \ldots\ \mid c_n \Rightarrow c_n[\boldsymbol{\tau}]\} \\ & \to_{\overline{\mu}} \ldots \end{aligned}$$

where $a_0 = a$ and

$$a_{i+1} = \mathsf{case}_{d[\boldsymbol{\tau}]}^{d[\boldsymbol{\tau}]}\ a_i\ \mathsf{of}\ \{c_1 \Rightarrow c_1[\boldsymbol{\tau}] \mid \ \ldots\ \mid c_n \Rightarrow c_n[\boldsymbol{\tau}]\}$$

Safe and Principled Language Interoperation *

Valery Trifonov and Zhong Shao

Department of Computer Science
Yale University
New Haven, CT 06520-8285
{trifonov-valery,shao-zhong}@cs.yale.edu

Abstract. Safety of interoperation of program fragments written in different safe languages may fail when the languages have different systems of computational effects: an exception raised by an ML function may have no valid semantic interpretation in the context of a Safe-C caller. Sandboxing costs performance and still may violate the semantics if effects are not taken into account. We show that effect annotations alone are insufficient to guarantee safety, and we present a type system with bounded effect polymorphism designed to verify the compatibility of abstract resources required by the computational models of the interoperating languages. The type system ensures single address space interoperability of statically typed languages with effect mechanisms built of modules for control and state. It is shown sound for safety with respect to the semantics of a language with constructs for selection, simulation, and blocking of resources, targeted as an intermediate language for optimization of resource handling.

1 Introduction

Component-based software development promises the freedom to choose the most suitable language independently for each fragment in a system, as long as the language implementation supports a common interface [12,15]. The existing interfaces offer a trade-off between safety and efficiency of interlanguage communication. The communicating programs may reside in separate address spaces, delegating the responsibility for correctness of their interaction to the operating system, which imposes severe performance penalties even for components using the same language implementation. Alternatively the caller and callee may share the same address space; this provides for fast interaction but typically fails to prevent possible errors due to inconsistency of the computational models. With the increasing dependence on component libraries the efficiency of the interoperation mechanism is becoming a significant factor; on the other hand the use of third-party components, especially in the context of dynamic linking, requires strong safety guarantees.

* This research was sponsored in part by the DARPA ITO under the title "Software Evolution using HOT Language Technology," DARPA Order No. D888, issued under Contract No. F30602-96-2-0232, and in part by an NSF CAREER Award CCR-9501624, and NSF Grant CCR-9633390. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

S.D. Swierstra (Ed.): ESOP/ETAPS'99, LNCS 1576, pp. 128–146, 2002.
© Springer-Verlag Berlin Heidelberg 2002

Interoperation between fragments with different computational models also occurs when a higher-level language is being used for systems programming. Demands for independence of some language features (e.g. garbage collection [11]) on parts of the system can be satisfied by compiling them using a different model for a subset of the language, and using an interlanguage protocol to communicate with the rest of the system. This approach can also be used when compiling finer-grained program fragments written in the same language, to achieve pay-as-you-go efficiency of language features using the most cost-effective model which supports the features employed by the particular fragment. On a higher level, different source languages may have a common representation in a typed intermediate language [16] and different but interoperating implementation schemes, tuned to specific source language characteristics. Safety and efficiency of interoperation are again definite but competing requirements in these situations.

To provide the basis for a safe and efficient interlanguage operation, in this paper we describe a novel type-based technique, supporting principled interoperation among languages with different features selected among mutable store, exceptions, first-class continuation, and heap and stack allocation of activation records. Our framework allows programs written in multiple languages with overlapping features to interact with each other safely and reliably, yet without restricting the expressiveness of each language.

Thus our goal is designing an interoperability scheme which is

- safe: it should not be possible to violate the runtime safety of a language by calling a foreign function, even if this function is defined in a language with different features; and
- efficient: a language implementation should not be forced to use suboptimal methods for its own features in order to provide support for other languages' features. For instance the implementation of a language that does not have exceptions should not have to know about the exception handling mechanism(s) used in interoperating implementations of other languages.

Ideally we would like to have complete interoperability, allowing us to invoke any function from any term written in another language as long as the semantics of this invocation is defined in the "union" of the languages. However this requirement poses serious efficiency problems, since to satisfy it, the implementation of each language should be aware of the supporting mechanisms for all features in the union language. Thus, for instance, if a Scheme implementation S employs heap-based allocation of activation records, an implementation of Safe-C which may have to interoperate with S cannot use a stack; or, conversely, the Scheme implementations will be forced to use an allocation strategy compatible with a stack [1,5].

At the other extreme are interfaces like COM [15] which impose few restrictions on language implementations. Safety is to be ensured via sandboxing, using separate address spaces. The interoperation mechanism supports only basic language features, e.g. function invocation and passing of arguments and result. In cases when the cost of cross-domain calls is acceptable this appears as a reasonable solution, but in fact depending on design choices in the implementations it may not provide the expected semantics.

ML:

```
exception E
fun callback () =... raise E ...
fun MLMain () =
  ...
  J.f (0,callback)
    handle E => J.f (1,callback)
  ...
```

Java:

```
class J {
  public static
    void f (int i, Callback c)
    throws Exception {
      if (i == 1) throw new Exception();
      try { c.invoke (); }
      catch (Exception e) { ... }
  }
}
```

Fig. 1. Failure of simple sandboxing to preserve semantics

Consider the schematic example shown in Figure 1, where raising exception `E` in `callback` shortcuts the flow through the Java fragment. If the Java implementation maintains information about the last entered `try` (i.e. the current exception handler) which is context-switched upon calls to ML, this information will be incorrect (with respect to the "union" semantics) when `Exception` is thrown after the second call to `J.f`.

Thus as observed in [14] the function of a mechanism for safe interlanguage calls is more than marshalling values between representations – it must also take into account the *effect systems* of the languages.

The contribution of this paper is that it formalizes the notion of safe interoperability between statically typed languages by building on previous work on effect systems [8,18,19] and introducing a type system which relates effects to the *machine resources* they require. Since different models of languages provide different sets of resources, and the same effect may be possible with various sets of resources, both an effect and a resource annotation are needed: the resource annotation indicates the specific requirements of a code fragment, while the effect annotation determines whether an alternative set of resources can be coerced to match these requirements.

Tracking the effects of the parameters of higher-order functions is achieved by effect polymorphism: using effect variables in the types to express the dependencies. More specifically our system has *bounded effect polymorphism* to reject effect applications when the effect arguments are unsupported by the resource bounds. We omit type polymorphism from the present description for brevity; we believe it is largely orthogonal to the treatment of resources and effects in types.

Furthermore, to show soundness of our type system, we introduce a typed language with constructs for explicit management of machine resources. Using this language as intermediate in compiling various source languages allows the compiler to optimize interlanguage calls by "floating" the boundary between contexts with different resource requirements. Thus parts of a program written in one language can be specialized to operate with the resources provided by the implementation of another language [17].

The result is that our system avoids the safety traps and allows for the interoperation of efficient implementations by restricting, in some cases, which foreign functions may be invoked, and imposing conditions the caller must satisfy before the invocation. To determine whether a call is possible, we consider the effects of a function, i.e. its use

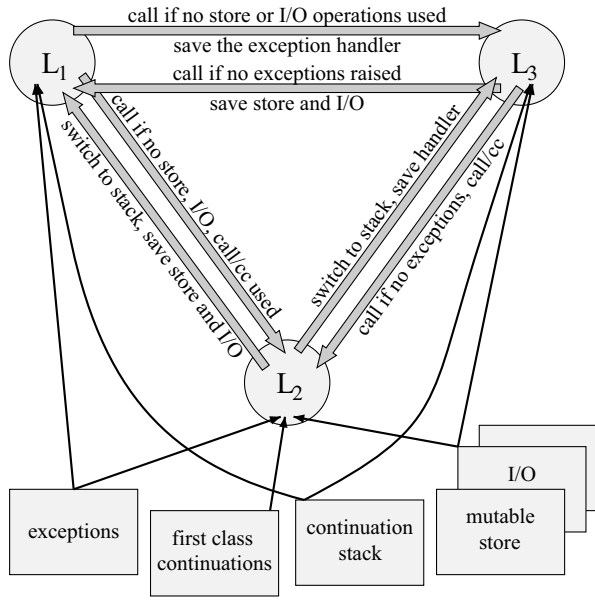


Fig. 2. Language interoperation: conditions for safety and efficiency

of resources. A resource has to be saved (blocked) when it is not required by the callee. If a resource which the caller does not have is needed by the called function, but the latter does not produce an effect depending on this resource, a dummy resource can be provided instead. In some cases it is possible to switch to alternative resources supporting the same effect. An example illustrating these points is shown in Figure 2 where three languages L_1, L_2, and L_3 are built out of the semantic modules studied in this paper. For example, calling an L_3 function from an L_2 program is always possible, because the functionality of all resources of L_3 is supported in L_2, but the calling convention must be switched from heap to stack based, and the L_2 exception handler must be preserved.

To specify formally and prove the safety of our system, in Section 2 we introduce the typed intermediate language $\mathcal{R}$ and describe its static and dynamic semantics in Section 3. In Section 3.3 we show that the type system of $\mathcal{R}$ guarantees the runtime safety of type-correct programs. This makes possible the safe linking of separately compiled components which can be shown to have a type-correct translation into $\mathcal{R}$ with the corresponding interface type.

2 A Language with Machine Resource Control

The language $\mathcal{R}$ is an idealized version of the typed intermediate language of our system. The novel feature in it are the *abstract resources*, which together with their associated primitive operations can be seen as modules of which we can build sublanguages of $\mathcal{R}$; indeed this is an implied goal of our interoperability scheme. Both the static and the dynamic semantics of $\mathcal{R}$ permit a presentation in which new functional blocks are

Abstract Resources

continuation

$$\mathit{ContRes} \ni a ::= \mathsf{S} \mid \mathsf{H} \qquad \text{continuation stack and heap}$$

control

$$\mathit{CtrlRes} \ni c ::= a \mid \mathsf{X} \qquad \ldots \text{ plus an exception handler}$$

primitive

$$\mathit{PrimRes} \ni r ::= c \mid \mathsf{M} \qquad \ldots \text{ plus a mutable store}$$

Resource Descriptors

$$\rho \in \mathit{ResourceDesc} = \mathit{CtrlRes} \times 2^{\mathit{PrimRes}}$$

Effects

$$u \in \mathit{EffVar}$$

$$\mathit{PrimEff} \ni f ::= \mathsf{callcc} \mid \mathsf{exception} \mid \mathsf{store}$$

$$\varepsilon \in \mathit{Effects} = 2^{\mathit{EffVar} \cup \mathit{PrimEff}}$$

Types

$$\mathit{Typ} \ni \tau ::= \tau \to^{\rho}_{\varepsilon} \tau \mid \mathsf{cont}^{\rho}[\tau] \mid \forall u \leq \rho.\, \tau \mid \mathsf{exn} \mid \mathsf{ref}[\tau] \mid \mathsf{unit} \mid b, \quad b \in \mathit{BasicTyp}$$

Values and Terms

$$\begin{array}{rll}
\mathit{Val} \ni v ::= & x \mid d & x \in \mathit{Var},\ d \in \mathit{Const} \supseteq \{*\} \\
\mid & \lambda^{\rho} x{:}\tau.\, e & \text{resource-specific abstractions} \\
\mid & \Lambda u \leq \rho.\, v & \text{bounded effect abstractions} \\
\mid & x[\varepsilon] & \text{effect applications} \\
\mathit{Exp} \ni e ::= & @\, x\, x' & \text{applications} \\
\mid & \mathbf{use}\,(\rho)\, e & \text{resource control} \\
\mid & \lfloor v \rfloor & \text{values} \\
\mid & \mathbf{let}\ x : \tau \leftarrow e\ \mathbf{in}\ e' & \text{bindings} \\
\mid & \mathbf{ref}\ x \mid\ !\, x \mid x := x' & \text{store} \\
\mid & \mathbf{callcc}\ x \mid \mathbf{throw}[\tau]\ x\ x' & \text{continuations} \\
\mid & e\ \mathbf{handle}\ x : \mathsf{exn}.\, e' \mid \mathbf{raise}[\tau]\ x & \text{exceptions}
\end{array}$$

Fig. 3. Syntax of $\mathcal{R}$, a language with typed resource control

added to a basic language without interference with other blocks; the exception is the exceptions block, whose semantics needs support from both first-class continuations (when present) and the resource handling itself. We take the approach of presenting all blocks in one step mainly due to space constraints.

The abstract resources, ranged over by r (see Figure 3), are structured in a hierarchy including the control resources c and their subdivision, the continuation allocation resources a. The language supports two allocation strategies for activation records: a stack-based discipline with abstract resource S, and a heap-based, with abstract resource H. An additional control resource is the exception handler X. Informally all of the control resources can be viewed as structures of frames such as activation records. A primitive resource not related to the control is the store M; the system can be directly extended with multiple versions of the store which can be controlled separately.

Primitive resources are the building blocks of the resource descriptors ρ, consisting of two components. The first component specifies the "calling convention" in use. i.e.

how values are communicated to and from a term; to keep the system simple we only consider conventions on returning the result, with a choice between a stack-allocated and a heap-allocated continuation. The second component describes the set of resources available or required for the evaluation of a term.

The counterpart of resources are the *effects* ε which are sets of primitive effects (informally caused by the corresponding primitive operations in the language) and effect variables which stand for sets of effects. In our language the primitive effects are callcc, exception, and store. A computation may only introduce effects which are provided for by the available resources, e.g. the effect exception can only occur when the exception handler resource X is available.

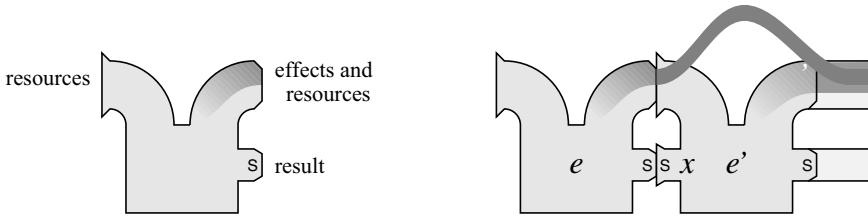
There are only minimal requirements for resources needed to produce an effect; extra resources can simply be ignored. This intuitive observation leads to the definition of a relation of compatibility between resource descriptors: letting rs range over sets of primitive resources, $\langle a, rs\rangle \sqsubseteq \langle a, rs'\rangle$ if $rs \subseteq rs'$. Note that compatible resource descriptors denote the same calling convention.

The types τ include function types $\tau_1 \to^{\rho}_{\varepsilon} \tau_2$ annotated with a resource descriptor ρ and effects ε. This notation describes a function whose evaluation requires the calling convention and resources denoted by ρ, and produces the effects ε. Similarly the resource annotation ρ on the type of continuations $\mathsf{cont}^{\rho}[\tau]$ denotes the resources needed to re-activate the continuation; the type system assumes that the effect of invoking a continuation is the maximal possible under ρ.

Among the types are also the bounded-effect quantified types $\forall u \le \rho.\, \tau$. An effect application $x[\varepsilon]$ of a variable x of this type is only valid when the effects ε are possible with the resources described in ρ.

In addition to effect applications the values v of $\mathcal{R}$ include bounded effect abstractions, and abstractions annotated with resource descriptors with the meaning noted for function types above.

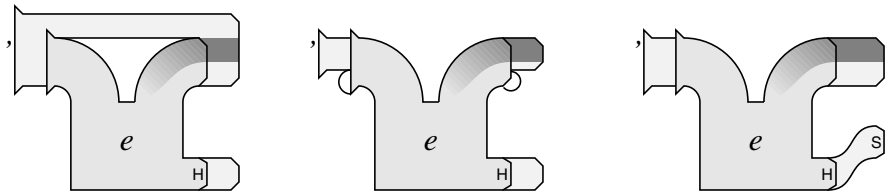
A term e which requires the resources described in ρ and produces effects ε can be visualized as the element shown in Figure 4(a), where the S indicates the convention for the result is to use a stack. The term $\lfloor v \rfloor$ denotes an effect-free computation returning the value v according to a convention determined by the context. The computation of **let** $x : \tau \leftarrow e$ **in** e' merges the effects of the computations of e and e' with x bound to the value of e (Figure 4(b)).



(a) A term requiring resources ρ and producing effects ε

(b) Evaluation of **let** $x : \tau \leftarrow e$ **in** e' with resources ρ

Fig. 4. Terms and their composition



(a) restricting the set of resources (b) extending (c) switching calling conventions

Fig. 5. Operations on resources performed by **use** $(\rho)\ e$

A novelty is the resource-management construct **use** $(\rho)\ e$, which evaluates e in the context of resources described by ρ, replacing the current resources. For instance, it is possible to reduce the set of available resources ρ' when a term e expects a subset ρ (as reflected in its type, see Section 3 for details). As alluded to in Figure 5(a), the resources currently available but not needed are preserved during the evaluation of e and restored upon completion. For example, consider the case of an ML program with resources $\langle \mathsf{H}, \{\mathsf{H}, \mathsf{X}, \mathsf{M}\}\rangle$ invoking a Scheme function f with an integer argument x and integer result. Assuming both implementations perform heap allocation of activation records, the call would require preserving the ML exception handler. Translated to $\mathcal{R}$, the invocation is expressed by

$$\textbf{use}\ (\langle \mathsf{H}, \{\mathsf{H}, \mathsf{M}\}\rangle)\ (@\ \mathsf{f}\ \mathsf{x})$$

in a type environment including $\mathsf{x} : \mathsf{Int}, \mathsf{f} : \mathsf{Int} \to^{\langle \mathsf{H},\{\mathsf{H},\mathsf{M}\}\rangle}_{\varepsilon} \mathsf{Int}$ (for some effect ε).

It is also useful to allow the creation of a new dummy resource r when it is required by a term e but the effects of e make no use of r. Consider a Scheme fragment with resources $\langle \mathsf{H}, \{\mathsf{H}, \mathsf{M}\}\rangle$ invoking a compiled from ML function g of type $\mathsf{Int} \to^{\langle \mathsf{H},\{\mathsf{H},\mathsf{X},\mathsf{M}\}\rangle}_{\{\mathsf{store}\}} \mathsf{Int}$. This type shows that g has no effects that require the exception handler, but the code for g nevertheless expects an exception handler resource, which is reflected in the resource component of g's type. Therefore the invocation must be enclosed in a **use**, constructing a dummy X resource:

$$\textbf{use}\ (\langle \mathsf{H}, \{\mathsf{H}, \mathsf{X}, \mathsf{M}\rangle)\ (@\ \mathsf{g}\ 5)$$

This situation is represented graphically in Figure 5(b) where the effects ε are supported by the resources ρ', but the term e requires additional resources.

In the absence of continuation capture effects the creation of new stack and heap resources has different semantics due to the localization of allocation effects on these resources. While a newly created store or exception handler resource cannot be used at all (since any use would create an effect which requires that resource, hence the term would not have a type in an environment which does not provide the resource), a new stack or heap may be used for the allocation of activation records, because the allocation effect is localized and will not be propagated when the evaluation of the term is completed. Note that this only applies to the use of the heap for stack-like management of activation records; use of **callcc** for instance introduces an effect which makes it impossible to type the term in an environment which does not have a heap resource.

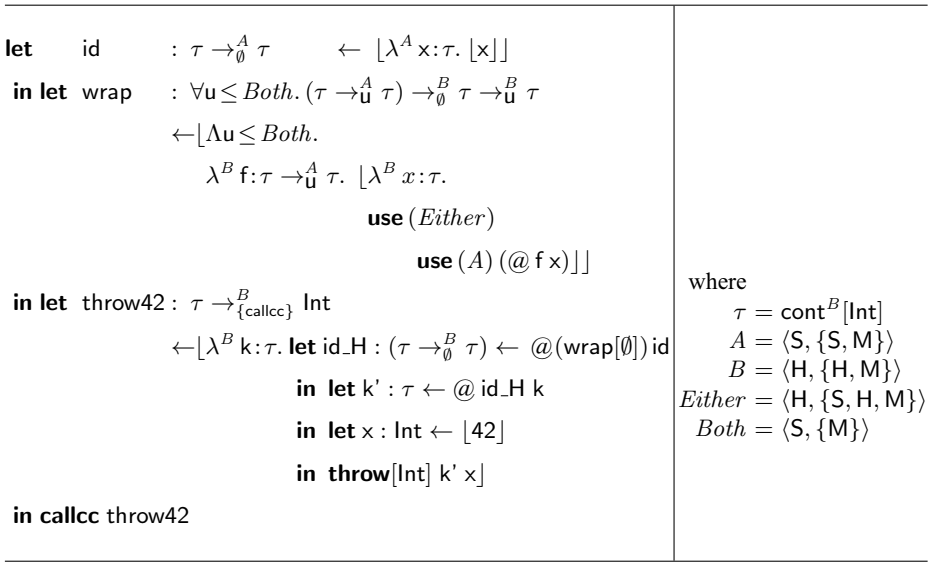


Fig. 6. Example of handling of foreign objects

Another application of the construct **use** $(\rho)\ e$ is the selection of calling convention. Figure 5(c) illustrates this in the case of switching from stack-based to heap-based continuations for the evaluation of e when $\rho = \langle \mathsf{H}, rs \rangle$ and the resource descriptor of the context is $\rho' = \langle \mathsf{S}, rs \rangle$, where $\{\mathsf{H}, \mathsf{S}\} \subseteq rs$ (i.e. both a heap and a stack are provided). Formal conditions for validity of **use** $(\rho)\ e$ are presented in Section 3.

The example in Figure 6 shows an application of **use** in the case of interoperation between programs written in two languages. The function id uses stack allocation (S), while throw42 uses heap allocation (H). Both languages also have the store resource M. Before id can be called from throw42, it must be coerced to heap allocation, which is performed by the effect-polymorphic function wrap. Note that the effects of the arguments of wrap are restricted by the resources in the intersection of the two languages' sets, denoted by $Both$; in this case this means only store effects are allowed. The invocation of wrap's argument f is enclosed in two **use** regions: the outer **use** creates a new stack, while the inner one switches the calling convention to the stack and saves the heap resource.

The example also shows how an object which only has meaning in a language with a given feature can be handled "passively" by code written in a language without support for that feature. Note that the first-class continuation captured in the heap-allocating program is passed to id and back, and then activated.

`Effect Environment Formation`

(Env-eff-empty)
$$\vdash_\Delta \emptyset$$

(Env-eff-ext)
$$\frac{\vdash_\Delta \Delta}{\vdash_\Delta \Delta_t,\ t \le \rho}$$

`Effects`

(Eff-empty)
$$\frac{\vdash_\Delta \Delta}{\Delta \vdash_\varepsilon \emptyset \le \rho}$$

(Eff-union)
$$\frac{\Delta \vdash_\varepsilon \varepsilon' \le \rho \quad \Delta \vdash_\varepsilon \varepsilon'' \le \rho}{\Delta \vdash_\varepsilon \varepsilon' \cup \varepsilon'' \le \rho}$$

(Eff-var)
$$\frac{\vdash_\Delta \Delta \quad u \in Dom(\Delta)}{\Delta \vdash_\varepsilon u \le \Delta(u)}$$

(Eff-primitive)
$$\frac{\vdash_\Delta \Delta \quad \rho \in Required(f)}{\Delta \vdash_\varepsilon \{f\} \le \rho}$$

(Eff-add-resource)
$$\frac{\Delta \vdash_\varepsilon \varepsilon \le \rho \quad \rho \sqsubseteq \rho'}{\Delta \vdash_\varepsilon \varepsilon \le \rho'}$$

`Types`

(Typ-basic)
$$\frac{\vdash_\Delta \Delta}{\Delta \vdash_\tau b}$$

(Typ-fun)
$$\frac{\Delta \vdash_\varepsilon \varepsilon \le \rho \quad \Delta \vdash_\tau \tau,\ \tau'}{\Delta \vdash_\tau \tau \to^\rho_\varepsilon \tau'}$$

(Typ-unit)
$$\frac{\vdash_\Delta \Delta}{\Delta \vdash_\tau \mathsf{unit}}$$

(Typ-poly)
$$\frac{\vdash_\Delta \Delta \quad \Delta_u,\ u \le \rho \vdash_\tau \tau}{\Delta \vdash_\tau \forall u \le \rho.\ \tau}$$

(Typ-ref)
$$\frac{\Delta \vdash_\tau \tau}{\Delta \vdash_\tau \mathsf{ref}[\tau]}$$

(Typ-cont)
$$\frac{\Delta \vdash_\tau \tau \quad \emptyset \vdash_\varepsilon \{\mathsf{callcc}\} \le \rho}{\Delta \vdash_\tau \mathsf{cont}^\rho[\tau]}$$

`Type Environment Formation`

(Env-typ-empty)
$$\frac{\vdash_\Delta \Delta}{\Delta \vdash_\Gamma \emptyset}$$

(Env-typ-ext)
$$\frac{\Delta \vdash_\Gamma \Gamma \quad \Delta \vdash_\tau \tau}{\Delta \vdash_\Gamma \Gamma_x,\ x : \tau}$$

`Values`

(Val-const)
$$\frac{\Delta \vdash_\Gamma \Gamma}{\Delta; \Gamma \vdash_v d : \theta(d)}$$

(Val-var)
$$\frac{\Delta \vdash_\Gamma \Gamma \quad x \in Dom(\Gamma)}{\Delta; \Gamma \vdash_v x : \Gamma(x)}$$

(Val-abs)
$$\frac{\Delta \vdash_\Gamma \Gamma \quad \Delta \vdash_\tau \tau \quad \rho; \Delta; \Gamma_x,\ x : \tau \vdash_e e : \tau'; \varepsilon}{\Delta; \Gamma \vdash_v \lambda^\rho x{:}\tau.\ e : \tau \to^\rho_\varepsilon \tau'}$$

(Val-eff-abs)
$$\frac{\Delta \vdash_\Gamma \Gamma \quad \Delta_u,\ u \le \rho; \Gamma \vdash_v v : \tau}{\Delta; \Gamma \vdash_v \Lambda u \le \rho.\ v : \forall u \le \rho.\ \tau}$$

(Val-eff-app)
$$\frac{\Gamma(x) = \forall u \le \rho.\ \tau \quad \Delta \vdash_\varepsilon \varepsilon \le \rho}{\Delta; \Gamma \vdash_v x[\varepsilon] : [\varepsilon/u]\tau}$$

Fig. 7. The $\mathcal{R}$ type system: effects, types and values

3 Semantics of $\mathcal{R}$

3.1 Static Semantics

The type system of $\mathcal{R}$, shown in Figures 7 and 8, keeps track of the resources necessary for the evaluation of a term and makes a conservative estimate of the effects of the evaluation. The effect environment Δ specifies the resource bounds of free effect variables, and as usual the type environment Γ assigns types to free variables.

The rules for sequents $\Delta \vdash_\varepsilon \varepsilon \le \rho$ reflect the dependence of effects on resources and form the basis of bounded effect polymorphism. The dependence of primitive effects on resources is captured by the function *Required* (Figure 9) specifying the alternatives for minimal resource descriptors enabling a primitive effect. Note that the exception and store effects work with either stack or heap continuation allocation, while the callcc effect can only be introduced with heap allocation.

(Exp-app)
$$\frac{\Delta \vdash_T \Gamma \quad \Gamma(x) = \tau' \to^{\rho}_{\varepsilon} \tau \quad \Gamma(x') = \tau'}{\rho; \Delta; \Gamma \vdash_e @\, x\ x' : \tau; \varepsilon}$$

(Exp-use)
$$\frac{\rho; \Delta; \Gamma \vdash_e e : \tau; \varepsilon \quad \Delta \vdash_\varepsilon \varepsilon \leq \rho'}{\rho'; \Delta; \Gamma \vdash_e \mathbf{use}\,(\rho)\, e : \tau; \varepsilon}$$

(Exp-val)
$$\frac{\Delta; \Gamma \vdash_v v : \tau}{\rho; \Delta; \Gamma \vdash_e \lfloor v \rfloor : \tau; \emptyset}$$

(Exp-let)
$$\frac{\rho; \Delta; \Gamma \vdash_e e : \tau; \varepsilon \quad \rho; \Delta; \Gamma_x,\, x : \tau \vdash_e e' : \tau'; \varepsilon'}{\rho; \Delta; \Gamma \vdash_e \mathbf{let}\ x : \tau \leftarrow e\ \mathbf{in}\ e' : \tau'; \varepsilon \cup \varepsilon'}$$

(Exp-callcc)
$$\frac{\Delta \vdash_T \Gamma \quad \Gamma(x) = \mathsf{cont}^{\rho}[\tau] \to^{\rho}_{\varepsilon} \tau}{\rho; \Delta; \Gamma \vdash_e \mathbf{callcc}\ x : \tau; \varepsilon \cup \{\mathsf{callcc}\}}$$

(Exp-throw)
$$\frac{\Delta \vdash_T \Gamma \quad \Delta \vdash_\tau \tau \quad \Gamma(x) = \mathsf{cont}^{\rho}[\tau'] \quad \Gamma(x') = \tau'}{\rho; \Delta; \Gamma \vdash_e \mathbf{throw}[\tau]\ x\ x' : \tau; \mathit{MaxEff}\ (\rho)} \quad \text{where } \mathit{MaxEff}\ (\rho) = \{f \mid \emptyset \vdash_\varepsilon \{f\} \leq \rho\}$$

(Exp-ref)
$$\frac{\Delta \vdash_T \Gamma \quad \Gamma(x) = \tau \quad \Delta \vdash_\varepsilon \{\mathsf{store}\} \leq \rho}{\rho; \Delta; \Gamma \vdash_e \mathbf{ref}\ x : \mathsf{ref}[\tau]; \{\mathsf{store}\}}$$

(Exp-deref)
$$\frac{\Delta \vdash_T \Gamma \quad \Gamma(x) = \mathsf{ref}[\tau] \quad \Delta \vdash_\varepsilon \{\mathsf{store}\} \leq \rho}{\rho; \Delta; \Gamma \vdash_e \mathbf{!}\ x : \tau; \{\mathsf{store}\}}$$

(Exp-update)
$$\frac{\Delta \vdash_T \Gamma \quad \Gamma(x) = \mathsf{ref}[\tau] \quad \Gamma(x') = \tau \quad \Delta \vdash_\varepsilon \{\mathsf{store}\} \leq \rho}{\rho; \Delta; \Gamma \vdash_e x \mathbf{:=} x' : \mathsf{unit}; \{\mathsf{store}\}}$$

(Exp-raise)
$$\frac{\Delta \vdash_T \Gamma \quad \Gamma(x) = \mathsf{exn} \quad \Delta \vdash_\varepsilon \{\mathsf{exception}\} \leq \rho}{\rho; \Delta; \Gamma \vdash_e \mathbf{raise}[\tau]\ x : \tau; \{\mathsf{exception}\}}$$

(Exp-handle)
$$\frac{\rho; \Delta; \Gamma \vdash_e e : \tau; \varepsilon \quad \rho; \Delta; \Gamma_x, x : \mathsf{exn} \vdash_e e' : \tau; \varepsilon' \quad \Delta \vdash_\varepsilon \{\mathsf{exception}\} \leq \rho}{\rho; \Delta; \Gamma \vdash_e e\ \mathbf{handle}\ x : \mathsf{exn}.\, e' : \tau; \varepsilon \cup \varepsilon' \cup \{\mathsf{exception}\}}$$

Fig. 8. The $\mathcal{R}$ type system: terms

Type judgments for values associate a type τ with a value v and a pair of environments; values have no effects and therefore their computation requires no resources. Sequents for terms have the form $\rho; \Delta; \Gamma \vdash_e e : \tau; \varepsilon$, where ρ is the resource descriptor of the environment, τ is the type of e, and ε represents the effects of the evaluation of e. For the typing of constants we assume the existence of a function $\theta \in \mathit{Const} \to \mathit{BasicTyp}$, and in particular that $\theta(*) = \mathsf{unit}$.

$$\begin{aligned} \mathit{Required}\,(\mathsf{callcc}) &= \{\langle \mathsf{H}, \{\mathsf{H}\}\rangle\} \\ \mathit{Required}\,(\mathsf{exception}) &= \{\langle \mathsf{S}, \{\mathsf{S}, \mathsf{X}\}\rangle,\ \langle \mathsf{H}, \{\mathsf{H}, \mathsf{X}\}\rangle\} \\ \mathit{Required}\,(\mathsf{store}) &= \{\langle \mathsf{S}, \{\mathsf{S}, \mathsf{M}\}\rangle,\ \langle \mathsf{H}, \{\mathsf{H}, \mathsf{M}\}\rangle\} \end{aligned}$$

Fig. 9. Minimal resource requirements for primitive effects

The function $MaxEff$ yields the maximal effect possible with the resources in ρ; it is used in making a conservative approximation of the effects of a continuation given the resources available at the point of its capture (in rule (Exp-throw)).

3.2 Dynamic Semantics

To separate the static and dynamic aspects of the use of resource descriptors we consider a variant of $\mathcal{R}$ with explicit resource annotations, where the abstract syntax for terms is

$$\begin{aligned} e ::= {} & @\, x\ x' \mid \mathbf{use}^{\rho}\,(\rho')\,e \\ \mid {} & \lfloor v \rfloor^{\rho} \mid \mathbf{let}^{\rho}\ x : \tau \leftarrow e\ \mathbf{in}\ e' \\ \mid {} & \mathbf{ref}^{\rho}\ x \mid \mathbf{!}^{\rho}\ x \mid x \mathbf{:=}^{\rho} x' \\ \mid {} & \mathbf{callcc}^{\rho}\ x \mid \mathbf{throw}^{\rho}[\tau]\ x\ x' \\ \mid {} & e\ \mathbf{handle}^{\rho}\ x : \mathsf{exn}.\ e' \mid \mathbf{raise}^{\rho}[\tau]\ x \end{aligned}$$

The translation of type-correct $\mathcal{R}$ terms into the annotated language is straightforward since the new annotations correspond to the resource descriptors ρ on the left of $\vdash_e$ in the typing sequents for those terms.

We present operational semantics of $\mathcal{R}$ following [3] in terms of a variant of the tail-call-safe $\mathrm{C}_a\mathrm{EK}$ machine. We prefer operational semantics because it allows us to show directly that reasonably efficient implementations exist. The original machine allocates an activation frame on the continuation stack when entering a **let**-binding (but not when entering a closure), and pops the frame to complete the binding. We extend the machine by including a component denoting additional machine resources, and by providing a heap-based alternative continuation allocation strategy supporting first-class continuations.

The transitions of the abstract machine are specified as a relation on machine configurations consisting of the term being evaluated, its environment, and the currently available machine resources; for the purposes of the proof of soundness of the type system we include in the configuration also the type of the evaluated term as well as an accumulator of effects. Having the annotations in the syntax makes it clear that the although the semantics of some constructs depend on what resources are available, this dependence can be resolved at compile time.

The semantic domains defining the meaning of the components of the abstract machine are listed in Figure 10. As usual the environment E maps variables to values converted to their internal representation as shown in Figure 11. Values are represented in the environment as closures with environments binding their free variables; no environment for effect variables is needed because effect instantiation is performed via substitution (Figure 11).

To assist with the proof of soundness of the type system we instrument the operational semantics to keep track of the current control resource, the accumulated effects of the evaluation, and the type of the term being evaluated. Further, the environment is extended to also record the type of each variable; type safety (Lemma 1) implies that all type annotations and tags only used for verification in the semantics may be erased without affecting the outcome of the evaluation of a correctly typed term. We use the shorthands ${}^{V}E(x)$ and ${}^{T}E(x)$ for the value and type components of $E(x)$, respectively, and we write

$$\begin{array}{r}\gamma\,(x,\,E) = {}^{V}E(x)\\ \gamma\,(x[\varepsilon],\,E) = \gamma\,([\varepsilon/u]v,\,E')\ \text{ if } {}^{V}E(x) = \mathsf{Closure}\ \langle \Lambda u \leq \rho.\, v,\, E'\rangle,\ \text{and } \emptyset \vdash_{\varepsilon} \varepsilon \leq \rho,\\ \gamma\,(v,\,E) = \mathsf{Closure}\ \langle v,\,E\rangle \text{ for other } v\end{array}$$

Fig. 11. Representation of values

$E[x \mapsto (w : \tau)]$ to denote the extension of E which assigns machine value w and type τ to x.

The modularity of the language blocks is supported in our framework by the representation of machine resources Rs as a record (tuple) with a tag $tag\,(Rs) \in Resources$ which yields the set of the corresponding primitive resources. The set of possible values of an individual machine resource corresponding to abstract resource r is denoted by $MR\,(r)$. Thus, assuming a canonical enumeration of resources, a tuple of resources with tag rs is an element of the singleton separated sum $\sum_{rs' \in \{rs\}} \prod_{r \in rs'} MR\,(r)$; we will denote this set by $MR^{T}\,(rs)$. The semantics make use of two families of total functions,

$$\begin{array}{l} inj^{r}_{rs-\{r\}} \in MR\,(r) \times MR^{T}\,(rs - \{r\}) \to MR^{T}\,(rs \cup \{r\})\\ proj^{r}_{rs \cup \{r\}} \in MR^{T}\,(rs \cup \{r\}) \to MR\,(r) \times MR^{T}\,(rs - \{r\}),\end{array}$$

indexed by a primitive abstract resource r and a tag rs, with the intention that $inj^{r}_{rs-\{r\}}$ maps $\langle R, Rs\rangle$ to the record containing R and all of Rs, and $proj^{r}_{rs\cup\{r\}}$ is its inverse. We also use $single^{r}_{rs} = \pi_1 \circ proj^{r}_{rs}$ and $drop^{r}_{rs} = \pi_2 \circ proj^{r}_{rs}$. The functions inj^{r}_{rs} and $inj^{r'}_{rs}$ (and similarly $proj$) are commutative for $r \neq r', \{r, r'\} \cap rs = \emptyset$; this commutativity allows the generalization of inj and $proj$ to the functions

$$\begin{array}{l} inj^{rs'}_{rs-rs'} \in MR^{T}(rs') \times MR^{T}(rs - rs') \to MR^{T}(rs \cup rs')\\ proj^{rs'}_{rs \cup rs'} \in MR^{T}(rs \cup rs') \to MR^{T}(rs') \times MR^{T}(rs - rs').\end{array}$$

Using these functions we can define natural liftings of operations linear in an individual resource R (i.e. in whose type R occurs exactly once positively in both the types of domain and codomain, or does not occur in the domain type and the codomain is linear in R) to operations on a record of resources Rs containing R: e.g. for $f^{r} \in S \times MR\,(r) \to S' \times MR\,(r)$ we define $f^{r}_{rs}\,(\in) S \times \prod^{T}_{r \in rs} MR\,(r) \to S' \times \prod^{T}_{r \in rs} MR\,(r)$ by

$$\begin{array}{r} f^{r}_{rs}\,(s, Rs) = \mathbf{let}\ \langle R, Rs'\rangle = proj^{r}_{rs}\,(Rs),\\ \langle s', R'\rangle = f^{r}(s, R)\\ \mathbf{in}\ \langle s', inj^{r}_{rs-\{r\}}\,(R', Rs')\rangle\end{array}$$

where $r \in rs$. With the generalized versions of inj and $proj$ this lifting extends to functions on records of resources as well.

For each primitive resource r there is an initial element $empty^{r}$ of the algebra of the corresponding machine resource; these elements are used in the simulation of resources (rule (add) in Figure 13).

A generic continuation machine resource C^{c} (corresponding to abstract control resource c) is described by the four stack operations $empty^{c}$, $isEmpty^{c}$, $newFrame^{c}$, and $topFrame^{c}$, where $topFrame^{c}$ is the inverse of $newFrame^{c}$ on non-empty C^{c}. We will

$$\begin{array}{l}\gamma\,(x, E) = {}^{V}E(x)\\ \gamma\,(x[\varepsilon],\ E) = \gamma\,([\varepsilon/u]v,\ E')\ \text{ if } {}^{V}E(x) = \mathsf{Closure}\ \langle \Lambda u \leq \rho.\, v,\ E'\rangle,\ \text{and } \emptyset \vdash_{\varepsilon} \varepsilon \leq \rho,\\ \gamma\,(v, E) = \mathsf{Closure}\ \langle v,\ E\rangle \text{ for other } v\end{array}$$

Fig. 11. Representation of values

$E[x \mapsto (w : \tau)]$ to denote the extension of E which assigns machine value w and type τ to x.

The modularity of the language blocks is supported in our framework by the representation of machine resources Rs as a record (tuple) with a tag $tag\,(Rs) \in Resources$ which yields the set of the corresponding primitive resources. The set of possible values of an individual machine resource corresponding to abstract resource r is denoted by $MR\,(r)$. Thus, assuming a canonical enumeration of resources, a tuple of resources with tag rs is an element of the singleton separated sum $\sum_{rs' \in \{rs\}} \prod_{r \in rs'} MR\,(r)$; we will denote this set by $MR^T\,(rs)$. The semantics make use of two families of total functions,

$$\begin{array}{l} inj^r_{rs-\{r\}} \in MR\,(r) \times MR^T\,(rs - \{r\}) \to MR^T\,(rs \cup \{r\})\\ proj^r_{rs \cup \{r\}} \in MR^T\,(rs \cup \{r\}) \to MR\,(r) \times MR^T\,(rs - \{r\}),\end{array}$$

indexed by a primitive abstract resource r and a tag rs, with the intention that $inj^r_{rs-\{r\}}$ maps $\langle R, Rs\rangle$ to the record containing R and all of Rs, and $proj^r_{rs\cup\{r\}}$ is its inverse. We also use $single^r_{rs} = \pi_1 \circ proj^r_{rs}$ and $drop^r_{rs} = \pi_2 \circ proj^r_{rs}$. The functions inj^r_{rs} and $inj^{r'}_{rs}$ (and similarly $proj$) are commutative for $r \neq r', \{r, r'\} \cap rs = \emptyset$; this commutativity allows the generalization of inj and $proj$ to the functions

$$\begin{array}{l} inj^{rs'}_{rs-rs'} \in MR^T(rs') \times MR^T(rs - rs') \to MR^T(rs \cup rs')\\ proj^{rs'}_{rs \cup rs'} \in MR^T(rs \cup rs') \to MR^T(rs') \times MR^T(rs - rs').\end{array}$$

Using these functions we can define natural liftings of operations linear in an individual resource R (i.e. in whose type R occurs exactly once positively in both the types of domain and codomain, or does not occur in the domain type and the codomain is linear in R) to operations on a record of resources Rs containing R: e.g. for $f^r \in S \times MR\,(r) \to S' \times MR\,(r)$ we define $f^r_{rs}\,(\in) S \times \prod^T_{r\in rs} MR\,(r) \to S' \times \prod^T_{r\in rs} MR\,(r)$ by

$$\begin{array}{rl} f^r_{rs}\,(s, Rs) = & \mathbf{let}\ \langle R, Rs'\rangle = proj^r_{rs}\,(Rs),\\ & \quad \langle s', R'\rangle = f^r(s, R)\\ & \mathbf{in}\ \langle s', inj^r_{rs-\{r\}}\,(R', Rs')\rangle\end{array}$$

where $r \in rs$. With the generalized versions of inj and $proj$ this lifting extends to functions on records of resources as well.

For each primitive resource r there is an initial element $empty^r$ of the algebra of the corresponding machine resource; these elements are used in the simulation of resources (rule (add) in Figure 13).

A generic continuation machine resource C^c (corresponding to abstract control resource c) is described by the four stack operations $empty^c$, $isEmpty^c$, $newFrame^c$, and $topFrame^c$, where $topFrame^c$ is the inverse of $newFrame^c$ on non-empty C^c. We will

use the more concise notation $F ::^{c}_{rs} Rs$ both for $newFrame^{c}_{rs}(F, Rs)$ and as a pattern standing for Rs' when $c \in rs$, $isEmpty^{c}_{rs}(Rs') = \mathsf{false}$, and $topFrame^{c}_{rs}(Rs') = \langle F, Rs\rangle$. A section $(F ::^{c}_{rs})$ stands for the function mapping Rs to $F ::^{c}_{rs} Rs$.

The stack-based implementation only provides the minimal functionality; the heap-based implementation can be used for non-sequential access as well. Exception handling uses a separate continuation stack.

Activation frames include a variant of the standard binding frames [3] to record the continuation completing a **let**-binding $\mathbf{let}^{\langle a,rs\rangle}\ x : \tau' \leftarrow e'\ \mathbf{in}\ e$ after evaluating e'. The binding frame is of the form $\mathsf{Bind}\ \langle \lambda^{\langle a,rs\rangle}\, x : \tau.\, e,\, E,\, \tau'\rangle$, where τ' is the type of e in environment ${}^{T}E$ extended with $x : \tau$, rs is the set of resources available for the evaluation of the **let**, and a is the allocation resource for the evaluation of e.

In addition there are two kinds of resource management continuation frames. A Restore frame indicates that a machine resource has been saved and removed from the current; a Drop frame signals that a dummy resource has been created and provided to code which only uses the it locally, or not at all (but has been compiled to expect it). The names of these frames suggest the operations that must be done to restore the status after the evaluation of the current term has completed, i.e. when the frame is encountered during unwinding the continuation.

The relation of computation $\mapsto_1$ on configurations is the union of the transition rules shown in Figures 12 and 13); the relation of computation is the reflexive transitive closure of $\mapsto_1$. The transition rules are grouped in classes based on the feature they implement as follows.

Building block	Rules	Figure
environment	(app), (let), (bind)	12
store	(ref), (deref), (update)	12
first-class continuations	(callcc), (throw)	13
exceptions	(handle), (raise)	13
resource management	(add), (remove), (redirect) (nop), (restore), (drop)	13

Notable among the transitions are those for resource management – they save a resource for future use and remove it from the currently available set, create a dummy resource, or select a different continuation. The interaction between resource management and exception handling is non-trivial because resources must be restored when exiting a **use**-region in any way. For that reason first-class continuations are restricted to accept only the resource set available at point of capture; unlike them, exception handlers are allocated on a stack, thus it is possible to allow exceptions not to be tied to specific resources by intercepting them and restoring the resources.

Our implementation of exceptions is more realistic than the typical [23] which propagates exception packets through all enclosing terms whose evaluations is pending. The exception handler forms a stack parallel to the continuation stack (or heap). Raising an exception (rule (raise)) creates an exception packet which is processed like a value with continuation on the exception handler stack; the bindings on this stack, created by **handle**, restore the default continuation. This scheme has on overhead if exceptions are not used, and overhead linear in number of catching handlers when an exception is thrown.

Environment

(app) $\langle @\, x_1\, x_2,\, E,\, a,\, Rs,\, \varepsilon,\, \tau\rangle \mapsto_1 \langle e',\, E'[x' \mapsto E(x_2)],\, a,\, Rs,\, \varepsilon,\, \tau\rangle$
where Closure $\langle \lambda^{\langle a,rs\rangle}\, x' : \tau'.\, e', E'\rangle = {}^{V}E(x_1)$
$\tau' = {}^{T}E(x_2)$

(let) $\langle \mathbf{let}^{\langle a,rs\rangle}\, x : \tau' \leftarrow e' \,\mathbf{in}\, e,\, E,\, a,\, Rs,\, \varepsilon,\, \tau\rangle \mapsto_1$
$\langle e',\, E,\, a,\, \mathsf{Bind}\, \langle \lambda^{\langle a,rs\rangle}\, x : \tau'.\, e, E|_{FV(e)-\{x\}}, \tau\rangle ::^{a}_{rs} Rs,\, \varepsilon,\, \tau'\rangle$

(bind) $\langle \lfloor v \rfloor^{\langle c,rs\rangle},\, E,\, c,\, \mathsf{Bind}\, \langle \lambda^{\langle a,rs\rangle}\, x : \tau.\, e', E', \tau'\rangle ::^{c}_{rs} Rs',\, \varepsilon,\, \tau\rangle \mapsto_1$
$\langle e',\, E'[x \mapsto (\gamma(v, E) : \tau)],\, a,\, Rs',\, \varepsilon,\, \tau'\rangle$

Store (rules valid when $\mathsf{M} \in rs$)

(ref) $\langle \mathbf{ref}^{\langle a,rs\rangle}\, x,\, E,\, a,\, Rs,\, \varepsilon,\, \tau\rangle \mapsto_1$
$\langle \lfloor x' \rfloor^{\langle a,rs\rangle},\, E[x' \mapsto (\mathsf{Loc}\, \ell : \tau)],\, a,\, Rs',\, \varepsilon \cup \{\mathsf{store}\},\, \tau\rangle$
where $\tau = \mathsf{ref}[{}^{T}E(x)]$
$\langle \ell, Rs'\rangle = ref^{\mathsf{M}}_{rs}\, ({}^{V}E(x),\, Rs)$
$x' \notin Dom\,(E)$

(deref) $\langle !^{\langle a,rs\rangle}\, x,\, E,\, a,\, Rs,\, \varepsilon,\, \tau\rangle \mapsto_1$
$\langle \lfloor x' \rfloor^{\langle a,rs\rangle},\, E[x' \mapsto (deref^{\mathsf{M}}_{rs}\, (\ell, Rs) : \tau)],\, a,\, Rs,\, \varepsilon \cup \{\mathsf{store}\},\, \tau\rangle$
where $(\mathsf{Loc}\, \ell : \mathsf{ref}[\tau]) = E(x)$
$x' \notin Dom\,(E)$

(update) $\langle x :=^{\langle a,rs\rangle}\, x',\, E,\, a,\, Rs,\, \varepsilon,\, \mathsf{unit}\rangle \mapsto_1$
$\langle \lfloor * \rfloor^{\langle a,rs\rangle},\, E,\, a,\, update^{\mathsf{M}}_{rs}\, (\langle \ell,\, {}^{V}E(x')\rangle, Rs),\, \varepsilon \cup \{\mathsf{store}\},\, \mathsf{unit}\rangle$
where $(\mathsf{Loc}\, \ell : \tau) = E(x)$
$\tau = \mathsf{ref}[{}^{T}E(x')]$

where $rs = tag\,(Rs),\, a \in rs$

Fig. 12. Instrumented transition rules, part 1

3.3 Soundness of the Type System

To prove soundness of the type system we extend it with rules for machine configurations, and prove that this extension has the subject reduction and progress properties. The interesting aspect of assigning a type to a configuration in this system is that some of the values in the environment may refer to resources which are currently inaccessible (blocked by an enclosing **use**), which complicates the notion of type correctness. The details of the proof are omitted for space considerations.

The progress and subject reduction properties are combined in the following lemma.

Lemma 1 (Safety). *If* $C = \langle e,\, E,\, c,\, Rs,\, \varepsilon,\, \tau\rangle$ *and* $\emptyset \vdash_C C : \tau_0; \varepsilon_0$*, then either* $e = \lfloor v \rfloor^{\langle c,rs\rangle}$ *for some value* v *such that* $\emptyset; {}^{T}E \vdash_v v : \tau$*, and* $isEmpty^{c}_{rs}\, (Rs) = \mathsf{true}$*, or there exists a configuration* C' *such that* $C \mapsto_1 C'$ *and* $\emptyset \vdash_C C' : \tau_0; \varepsilon_0$.

(Note that the case of a value in an empty continuation covers both normal termination of the program and the case of an unhandled exception propagated to the top.)

As a corollary we obtain soundness of the system.

`First-class Continuations` (rules valid when $\mathsf{H} \in rs$)

(callcc) $\langle \mathbf{callcc}^{\langle \mathsf{H}, rs\rangle}\ x,\ E,\ Rs,\ \mathsf{H},\ \varepsilon,\ \tau\rangle \ \mapsto_1$
$\langle e',\ E'[x' \mapsto (\mathsf{Cont}\ k : \mathsf{cont}^{\langle \mathsf{H}, rs\rangle}[\tau])],\ \mathsf{H},\ Rs',\ \varepsilon \cup \{\mathsf{callcc}\},\ \tau\rangle$
where $\langle k, K, k_e\rangle = single^{\mathsf{H}}_{rs}\ (Rs)$
$Rs' = \mathbf{if}\ \mathsf{X} \notin rs\ \mathbf{then}\ Rs\ \mathbf{else}\ restoreExn\ (\mathsf{H})(Rs)$
$^{V}E(x) = \mathsf{Closure}\ \langle \lambda^{\langle \mathsf{H}, rs\rangle}\ x' : \mathsf{cont}^{\langle \mathsf{H}, rs\rangle}[\tau].\ e', E'\rangle$

(throw) $\langle \mathbf{throw}^{\langle \mathsf{H}, rs\rangle}[\tau]\ x_1\ x_2,\ E,\ a,\ Rs,\ \varepsilon,\ \tau\rangle \ \mapsto_1$
$\langle \lfloor x_2 \rfloor^{\langle \mathsf{H}, rs\rangle},\ E,\ \mathsf{H},\ inj^{\mathsf{H}}_{rs-\{\mathsf{H}\}}\ (\langle k_1, K, k_e\rangle,\ Rs'),\ \varepsilon,\ \tau''\rangle$
where $(\mathsf{Cont}\ k_1 : \mathsf{cont}^{\langle \mathsf{H}, rs\rangle}[\tau'']) = E(x_1)$
$\langle \langle k, K, k_e\rangle,\ Rs'\rangle = proj^{\mathsf{H}}_{rs}\ (Rs)$

`Exceptions` (rules valid when $\mathsf{X} \in rs$)

(handle) $\langle e\ \mathbf{handle}^{\langle a, rs\rangle}\ x : \mathsf{exn}.\ e',\ E,\ a,\ Rs,\ \varepsilon,\ \tau\rangle \ \mapsto_1\ \langle e,\ E,\ a,\ Rs',\ \varepsilon \cup \{\mathsf{exception}\},\ \tau\rangle$
where $Rs' = \left(\begin{array}{l} restoreCont\ (a, Rs) \circ \\ (\mathsf{Bind}\ \langle \lambda^{\langle a, rs\rangle}\ x : \mathsf{exn}.\ e', E|_{FV\,(e')-\{x\}}, \tau\rangle ::^{\mathsf{X}}_{rs})\ \circ \\ restoreExn\ (a) \end{array}\right)\ (Rs)$

(raise) $\langle \mathbf{raise}^{\langle a, rs\rangle}[\tau]\ x,\ E,\ a,\ Rs,\ \varepsilon,\ \tau\rangle \ \mapsto_1\ \langle \lfloor x \rfloor^{\langle \mathsf{X}, rs\rangle},\ E,\ \mathsf{X},\ Rs,\ \varepsilon \cup \{\mathsf{exception}\},\ \mathsf{exn}\rangle$

`Resource Management`

(add) $\langle \mathbf{use}^{\langle a, rs\rangle}\ (\langle a, rs'\rangle)\ e,\ E,\ a,\ Rs,\ \varepsilon,\ \tau\rangle \ \mapsto_1$
$\langle \mathbf{use}^{\langle a, rs \cup \{r\}\rangle}\ (\langle a, rs'\rangle)\ e,\ E,\ a,\ inj^{r}_{rs}\ (empty^{r}, \mathsf{Drop}\ r ::^{a}_{rs} Rs'),\ \varepsilon,\ \tau\rangle$
where $r \in rs' - rs$
$Rs' = \mathbf{if}\ \mathsf{X} \notin rs\ \mathbf{then}\ Rs\ \mathbf{else}\ \mathsf{Drop}\ r ::^{\mathsf{X}}_{rs} restoreExn\ (a)(Rs)$

(block) $\langle \mathbf{use}^{\langle a, rs\rangle}\ (\langle a, rs'\rangle)\ e,\ E,\ a,\ Rs,\ \varepsilon,\ \tau\rangle \ \mapsto_1$
$\langle \mathbf{use}^{\langle a, rs - \{r\}\rangle}\ (\langle a, rs'\rangle)\ e,\ E,\ a,\ \mathsf{Restore}\ \langle r, R\rangle ::^{a}_{rs} Rs'',\ \varepsilon,\ \tau\rangle$
where $r \in rs - rs' - \{a\}$
$Rs'' = \mathbf{if}\ \mathsf{X} \notin rs'\ \mathbf{then}\ Rs\ \mathbf{else}\ \mathsf{Restore}\ \langle r, R\rangle ::^{\mathsf{X}}_{rs} restoreExn\ (a)(Rs)$
$\langle R, Rs'\rangle = proj^{r}_{rs}\ (Rs)$

(redirect) $\langle \mathbf{use}^{\langle a, rs\rangle}\ (\langle a', rs'\rangle)\ e,\ E,\ a,\ Rs,\ \varepsilon,\ \tau\rangle \ \mapsto_1$
$\langle \mathbf{use}^{\langle a', rs\rangle}\ (\langle a', rs'\rangle)\ e,\ E,\ a',\ \mathsf{Bind}\ \langle \lambda^{\langle a, rs\rangle}\ x : \tau.\ \lfloor x \rfloor^{\langle a, rs\rangle},\ E,\ \tau\rangle ::^{a'}_{rs} Rs,\ \varepsilon,\ \tau\rangle$

(nop) $\langle \mathbf{use}^{\langle a, rs\rangle}\ (\langle a, rs\rangle)\ e,\ E,\ a,\ Rs,\ \varepsilon,\ \tau\rangle \ \mapsto_1\ \langle e,\ E,\ a,\ Rs,\ \varepsilon,\ \tau\rangle$

(restore) $\langle \lfloor v \rfloor^{\langle c, rs\rangle},\ E,\ c,\ \mathsf{Restore}\ \langle r, R\rangle ::^{c}_{rs} Rs',\ \varepsilon,\ \tau\rangle \ \mapsto_1$
$\langle \lfloor v \rfloor^{\langle c, rs\rangle},\ E,\ c,\ inj^{r}_{rs}\ (R,\ Rs'),\ \varepsilon,\ \tau\rangle$
where $c \in rs,\ r \notin rs$

(drop) $\langle \lfloor v \rfloor^{\langle c, rs\rangle},\ E,\ c,\ \mathsf{Drop}\ r ::^{c}_{rs} Rs',\ \varepsilon,\ \tau\rangle \ \mapsto_1\ \langle \lfloor v \rfloor^{\langle c, rs\rangle},\ E,\ c,\ drop^{r}_{rs}\ (Rs'),\ \varepsilon,\ \tau\rangle$
where $c \in rs,\ r \in rs - \{c\}$

where
$rs = tag\ (Rs)$
$a \in rs$
$restoreExn\ (a)(Rs) = replaceResource\ (a, \mathsf{X}, Rs)(Rs)$
$restoreCont\ (a, Rs') = replaceResource\ (\mathsf{X}, a, Rs')$
$replaceResource\ (c, r, Rs') = (\mathsf{Drop}\ r ::^{c}_{rs}) \circ (\mathsf{Restore}\ \langle r, single^{r}_{rs}\ (Rs')\rangle ::^{c}_{rs})$
where $tag\ (Rs') = rs,\ \{c, r\} \subseteq rs$

Fig. 13. Instrumented transition rules, part 2

Theorem 1 (Soundness). *If* $C = \langle e, E, c, Rs, \varepsilon, \tau \rangle$ *and* $\emptyset \vdash_C C : \tau_0; \varepsilon_0$*, then* C *computes to a value, or to an exception packet, or its computation diverges.*

4 Related Work

Interoperability is a primary concern of component-based models such as CORBA [12] and Microsoft's COM [15,13]. Safety in these systems is in conflict with the performance requirements, and even sandboxing may fail to provide correct semantics when the effect systems of the communicating languages go beyond the value/store interface. In comparison our system allows flexible and efficient interoperation between languages with different resources, and ensures safety by exposing the resource and effect requirements in the types of the components.

Foreign function call interfaces [6,14] are related in purpose but are designed under the constraint to be compatible with legacy code which is often unsafe. Solutions to this problem have major impact on interoperability today. We do not attempt to solve compatibility issues in the system presented in this paper. Our design is for interoperability between safe components with language-independent interfaces, aimed to satisfy high performance requirements when running in shared address space. We emphasize building a *safe*, *efficient*, and *robust* interface across multiple HOT languages.

The present work extends our earlier results [17] on a type system with effect and resource control for continuation allocation. While the state resource is essentially independent of the rest, the interactions of the exception resource with the continuation resources are non-trivial.

Although we do not present the semantics in terms of monads, the idea to use both resources and effects to describe a function's interoperability was inspired by recent work on monad-based interactions and modular interpreters [21,9,20,10], and Wadler's work on the relationship between monads and effects [22]. Monads, viewed as compositions of basic monad transformers [10], can be used to represent sets of resources. The transition rules creating a binding activation frame and binding a variable to a value (Figure 12), given a set of resources, provide the semantics of the `‘bind‘` and `unit` of the monad. Resource-specific primitives are interpreted by lifting the operations of the corresponding monad through the monad transformers enclosing it in the composition.

What makes our approach different is that our system keeps track of effects, which allows us to determine which components of a monad transformer composition are being used only trivially (i.e. only their `‘bind‘` and `unit` are invoked) and therefore can be eliminated or simulated. Furthermore, we wish to extend and reduce the set of resources in a commutative way, and for that purpose we represent the result of transformer compositions "horizontally" – the corresponding resources are collected in one component of the abstract machine configuration. Thus if the set of resources required by term e corresponds to the monad $M = (T_1 \circ \ldots \circ T_n)Id$, extending it via $\mathbf{use}^{\rho}\,(e)$ is, in general, not expressed as an application of a monad transformer T to M, but as the use of a monad morphism [2] to embed values of M into a monad isomorphic to a composition of $T_1, \ldots, T_n, T$ in a canonical order. Proving the equivalence of a monads defined as compositions with their horizontal representations meets the technical complexity of constructing morphisms between them. We have opted instead for operational seman-

tics which gives a more direct correspondence with an implementation. In our semantics the complexity of lifting monads through monad transformers is reflected in the interaction between various resources, e.g. in transition rules (callcc), (add), and (block) (Figure 13).

Closely related to our work is the research on effect systems [4,7,8,18,19]. The effects in our system are used for verifying interoperability constraints; in this context the novel bounded effect quantification is introduced as a form of effect polymorphism under resource restrictions which allows us to take advantage of the effect-resource relationship to support advanced compilation strategies. The effect inference suggested in the typing rules in Figure 8 is conservative and there is considerable room for improvement borrowing from prior work by e.g. finer separation of effects and adopting region inference or for determining their localization; however it is still not clear to what extent this improvement will materialize in practice – for instance the exception effects can be easily localized to a handler, but due to the typical extensibility of the exception type most handlers re-raise exceptions. We intend to test these variations in a prototype implementation under development within the FLINT system.

Acknowledgment

We thank the anonymous referees for suggestions on improving the presentation.

References

1. William D Clinger, Anne H Hartheimer, and Eric M Ost. Implementation strategies for continuations. In *ACM Conference on Lisp and Functional Programming*, pages 124–131, New York, June 1988. ACM Press.
2. Andrzej Filinski. *Controlling Effects*. PhD thesis, CMU, 1996.
3. Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 237–247, New York, June 1993. ACM Press.
4. David K. Gifford, Pierre Jouvelot, John M. Lucassen, and Mark A. Sheldon. FX-87 reference manual. Technical Report MIT/LCS/TR-407, M.I.T. Laboratory for Computer Science, September 1987.
5. Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 66–77, New York, 1990. ACM Press.
6. Lorenz Huelsbergen. A portable C interface for Standard ML of New Jersey. Technical memorandum, AT&T Bell Laboratories, Murray Hill, NJ, January 1996.
7. Pierre Jouvelot and David K. Gifford. Reasoning about continuations with control effects. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 218–226. ACM Press, 1989.
8. Pierre Jouvelot and David K. Gifford. Algebraic reconstruction of types and effects. In *18th Annual ACM Symposium on Principles of Programming Languages*, pages 303–310, New York, Jan 1991. ACM Press.
9. John Launchbury and Simon Peyton Jones. Lazy functional state threads. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 24–35, New York, June 1994. ACM Press.

10. Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 333–343. ACM Press, 1995.
11. Greg Nelson, editor. *Systems programming with Modula-3*. Prentice Hall, Englewood Cliffs, NJ, 1991.
12. Object Management Group (OMG). The Common Object Request Broker: Architecture and specifications (CORBA). Revision 1.2., Object Management Group (OMG), Framingham, MA, December 1993.
13. Simon Peyton Jones, Eric Meijer, and Daan Leijen. Scripting COM components in Haskell. Available at http://www.dcs.gla.ac.uk/~simonpj/com.ps.gz, 1997.
14. Simon Peyton Jones, Thomas Nordin, and Alastair Reid. Green card: A foreign-language interface for Haskell. Available at http://www.dcs.gla.ac.uk/fp/authors/Simon_Peyton_Jones/green-card-1.ps.gz, 1997.
15. Dale Rogerson. *Inside COM: Microsoft's Component Object Model*. Microsoft Press, 1997.
16. Zhong Shao. Typed common intermediate format. In *USENIX Conference on Domain Specific Languages*, pages 89–102, October 1997.
17. Zhong Shao and Valery Trifonov. Type-directed continuation allocation. In *2nd International Workshop on Types in Compilation*, volume 1473 of *LNCS*, pages 116–135, Berlin, 1998. Springer.
18. Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region, and effect inference. *Journal of Functional Programming*, 2(3), 1992.
19. Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and Computation*, 111(2):245–296, June 1994.
20. Philip Wadler. The essence of functional programming (invited talk). In *19th Annual ACM Symposium on Principles of Programming Languages*, New York, Jan 1992. ACM Press.
21. Philip Wadler. How to declare an imperative (invited talk). In *International Logic Programming Symposium*, Portland, Oregon, December 1995. MIT Press.
22. Philip Wadler. The marriage of effects and monads. In *ACM SIGPLAN International Conference on Functional Programming*, pages 63–74. ACM Press, 1998.
23. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.

Deterministic Expressions in C

Michael Norrish

Computer Laboratory
University of Cambridge**
mn200@cl.cam.ac.uk

Abstract. Expressions in the programming language C have such an under-specified semantics that one might expect them to be non-deterministic. However, with the help of a mechanised formalisation, we have shown that the semantics' additional constraints actually result in a large class of C expressions having only one possible behaviour.

1 Introduction

The semantics of the programming language C is specified in an ISO standard [3]. However, this semantics is written in natural language, and is thus unsuitable as the basis for formal work such as verification. Indeed, there are a number of unresolved disputes about various details in this standard.[1]

However, our Cholera formalisation [6,7] is a completely formal semantics for the bulk of the C language. It is formulated in a structural operational style (see, for example, [2]) and is embedded in the HOL theorem prover [1]. On this basis, it *is* possible to prove facts about the C language (modulo the degree of certainty with which one believes the formalisation to be correct). For example, it is possible to derive various "axiomatic" rules that allow one to reason about C programs with Hoare-like triples, as described in [5,7].

The work described here considers the semantics of C expressions, and in particular demonstrates that a significant class of these expressions are deterministic. This is an important result in the context of verification because it allows one to perform a verification with respect to just one possible path of execution. Otherwise, if an expression can evaluate in n different ways, then any verification of a program that contains it must demonstrate that the final post-condition holds for all n possibilities, a tedious task at best.

** Fax: +44 1223 334678

[1] See for example the Usenet newsgroup `comp.std.c`, where issues such as whether or not function calls may interleave are debated.

S.D. Swierstra (Ed.): ESOP/ETAPS'99, LNCS 1576, pp. 147–161, 1999.
© Springer-Verlag Berlin Heidelberg 1999

In addition to defining the semantics of C, our Cholera project aims to put results like this determinism theorem to work: using them in the verification of not entirely trivial programming examples. Such verification examples will support the thesis that verification of programs in complicated programming languages is possible, particularly if one has mechanical support for the task. Moreover, the fact that proofs of this nature are practical is an indication that even programming language semantics can be satisfactorily mechanised.

The remainder of this paper will first describe the relevant parts of the C semantics in section 2. In section 3, we explain why what might initially appear to be non-deterministic is in fact deterministic, and outline the overall proof strategy. The proof is explained in more detail in sections 4 and 5, and section 6 concludes.

2 The semantics of C expressions

Cholera models the semantics of C's expressions with a reduction style operational semantics using a relation $\rightarrow_e$ such that $\langle e_0, \sigma_0\rangle \rightarrow_e \langle e, \sigma\rangle$ holds when an expression e_0 in state σ_0 can take a step, becoming a new expression e, and with the state changing to state σ. States σ, σ_0, σ' etc. embody not just the usual mapping from variables to values, but also information about the program environment, and pending side effects. We use $\rightarrow_e^*$ to denote the reflexive and transitive closure of the single step relation.

There are two principal sources of non-determinism in the semantics. These are the rules for the evaluation of binary expressions and the way in which side effects are applied. The following two rules illustrate the first:

$$\frac{\langle e_1, \sigma_0\rangle \rightarrow_e \langle e, \sigma\rangle}{\langle e_1 \odot e_2, \sigma_0\rangle \rightarrow_e \langle e \odot e_2, \sigma\rangle} \qquad \frac{\langle e_2, \sigma_0\rangle \rightarrow_e \langle e, \sigma\rangle}{\langle e_1 \odot e_2, \sigma_0\rangle \rightarrow_e \langle e_1 \odot e, \sigma\rangle}$$

Here $\odot$ stands for all of C's arithmetic operators, but not for the logical operators `&&` and `||`, nor the comma operator, nor the assignment operators. The first rule says that if the first argument to a binary operator can take a step in the semantics, then so too can the containing expression. The non-determinism enters because both rules apply at all times, meaning that an expression is *not* constrained to evaluate its operands in any particular order, and may even interleave the evaluation of its operands. In the presence of side effects and changes to the state, this is potentially a significant source of non-determinism.

The second source of non-determinism in the semantics is its handling of side effects. Side effects are generated by the evaluation of assignment expressions and the various increment and decrement operators (`++` and `--`). These operators have as their side effects the writing of values into memory, but this does not necessarily happen immediately. Instead, side effects are applied at arbitrary times and in any order, subject only to the constraint that all pending side effects be applied before the next *sequence point*. Sequence points occur at certain well-marked stages in expression evaluation, such as after the complete evaluation of the first argument to the logical operators `&&` and `||`. The rule for side effect application is:

$$\frac{\eta \text{ is pending in } \sigma}{\langle e, \sigma\rangle \rightarrow_e \langle e, \texttt{apply\_se}(\sigma, \eta)\rangle}$$

where $\texttt{apply\_se}(\sigma, \eta)$ denotes the state resulting from the application of side effect η to state σ, with the appropriate changes made (memory updated, and η removed from the pending side effects).

2.1 Constraints on expression evaluation

The above description of expression evaluation suggests a chaotic picture. A naive interpretation would suggest that the evaluation of

```
v + v++ + v + v++
```

with `v` initally 3, could yield any value in the range 12–17. However, the language definition imposes severe constraints on the way in which expressions can evaluate, and in fact, this expression is undefined.

The constraint is that "between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be accessed only to determine the value to be stored." [3, §6.3] (For our purposes, an object is best understood as simply a part of memory.) Violation of this constraint results in undefinedness.

It is worth noting that this is a constraint on the dynamic behaviour of the program. Though the expression given above involving `v` will necessarily be undefined because it both refers to and updates the object denoted by `v`, it is not clear whether or not this is true of `*p + (i = 1)`, say, as it is impossible in general to determine whether or not `*p` (a dereferencing of pointer variable `p`) will refer to `i`. Finally, note that the second sentence quoted above allows references to take place if they occur on the right-hand of an assignment expression and the references are to

the object being modified by the assignment. For example, this clause allows `i = i + 1`.

A formal semantics of C must model this constraint as well as the more obvious rules given earlier. To do this, Cholera keeps track of three state components:

- the pending side effects
- those parts of memory which have been updated (the "`update_map`")
- those parts of memory which have been referred to (the "`ref_map`")

The pending side effects component is a multi-set, or bag, as the same side effect might occur twice in a given evaluation. The `update_map` is a set of addresses, as no evaluation will be allowed to update the same location twice. The `ref_map` is another bag, as multiple references to the same location can legitimately occur. As we shall see, we need to know how many references were made to a particular location, not just whether or not something has been referred to.

There are four different ways in which these components can change in the evaluation of an expression.

- When a non-array lvalue becomes a value, the `ref_map` is increased to reflect the reference of the object denoted by the lvalue.[2] If the part of memory referred to is in the `update_map`, this causes undefinedness.
- When a side effect is applied, it is removed from the pending side effects bag, and the `update_map` is increased, recording the fact that part of memory has just been changed. If that part of memory has already been updated or referred to, this causes undefinedness.
- When an assignment completes its evaluation, a side effect to update the appropriate part of memory with a new value is added to the pending side effects bag. Assignment expressions keep track of references made on their right hand sides, and those that were to the object to be updated are removed from the `ref_map`. Failure to do this would cause the side effect created as a result of evaluating `i = i + 1` to clash with the reference to `i` on the expression's RHS.
  Because the `ref_map` records a count of the number of times a piece of memory has been referred to, this deletion of references may still leave references recorded. Using only a set for `ref_map` would allow

```
i + (i = i + 1)
```

[2] Array lvalues become pointers to their first element; this transformation does not require a reference to memory.

to avoid revealing its undefined nature. A possible evaluation would have the `i` on the assignment's RHS remove the record of a previous reference to `i` on the LHS of the addition.
- When the pending side effects bag is empty, and a sequence point is reached in an expression's syntax, the `ref_map` and `update_map` are "zero-ed", thereby allowing a new sequence of reference and updates in the next phase of execution. If a sequence point is reached, and the bag of pending side effects is not empty, it will need to be emptied before the next stage of the expression can be evaluated.

3 Intuition and proof outline

This may have already suggested that C's expression semantics, though superficially full of non-determinism, is actually so seriously constrained that expressions can only evaluate in one way, whether this be to one valid result, or to undefinedness. Here we suggest why this is the case, and sketch the form of the proof that is to come.

Ignoring for the moment the fact that side effects are not necessarily applied immediately nor in order, one can think of the various sub-expressions of a greater expression as parallel processes running simultaneously and sharing memory. Clearly, the behaviour of these processes is solely dependent on the parts of memory that they reference. But this implies that the processes can't affect each other: a change to a piece of memory by one process that another references is forbidden by the constraints spelled out in the previous section.

If a sub-expression can't affect the parts of memory that another depends on, and *vice versa*, then their evaluation must proceed entirely deterministically. Conversely, if shared memory *is* updated illegally, then undefinedness must result.

The fact that side effects are not applied immediately is also seen to be irrelevant. All side effects will come to be applied eventually, as reaching a sequence point requires this, and at the minimum, there is a sequence point at the end of the evaluation of all expressions that appear within statements. Though an update may come quite late, the constraints forbid the updating of memory that has been referred to as much as they forbid reference of updated memory.

Nonetheless, there is still a problem with the above intuition: it ignores the effect of sequence points that appear within an expression. Consider the following expression:

```
x + ((x = 3), 4)
```

Given what we have seen so far of the semantics, it would appear that this expression should be genuinely non-deterministic. The comma operator on the right is a sequence point, so if an evaluation were to proceed by first evaluating `x = 3`, reaching the sequence point, clearing the `update_map`, and then proceeding with the rest of the expression, it should go on to give a result of 7.

On the other hand, if the lone `x` on the left were to be evaluated first, then the subsequent assignment expression (necessarily the next thing to be evaluated) would cause undefinedness, because it would update an object which had already been referenced.

In fact, a subtle argument about this case forces the conclusion that the expression is necessarily undefined.[3] An official response by the Standards committee to a public query (a "Defect report") [4, #117] makes it clear that if it is possible for an expression to exhibit undefined behaviour (there might be an order of evaluation that does this, for example), then the whole expression *is* undefined.

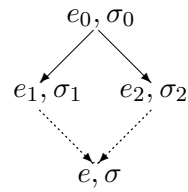
Cholera does model this requirement, but is forced to do so at the level above the definition of $\rightarrow_e$. We add a rule to the effect that a given, defined, reduction sequence is only part of the semantics if there doesn't exist any other sequence which makes the behaviour undefined. However, this additional detail in the semantics is difficult to reason about, so we choose to examine those expressions which are free of internal sequence points. Unless otherwise stated, all results stated here will be for expressions that are free of internal sequence points.

Our eventual determinism result for sequence point free expressions then naturally holds of expressions where the only sequence points are present at the top level (such as in `x || (y && z)`). This is because such an expression has deterministic sub-expressions, and these must be evaluated in the order dictated by the presence of the sequence points, giving an overall behaviour which must also be deterministic.

3.1 Proof outline

We should like to demonstrate determinism by showing a *diamond property* for all of the possible reductions that an expression might undergo. Graphically, this amounts to showing that in all situations we can find reductions to fill in the dashed lines below:

[3] My thanks to Mark Brader for explaining this to me.



It follows that if this can be shown for single steps of a reduction relation, then that reduction system must be confluent. Unfortunately, this property does *not* hold in general for C. In particular, reductions that involve undefined behaviour tend to invalidate further reductions, and if the reduction to $\langle e_1, \sigma_1 \rangle$, say, caused undefined behaviour then there is no guarantee that a reduction analogous to the one taken to get to $\langle e_2, \sigma_2 \rangle$ should be possible.

Therefore, the first step in attacking the proof is to divide it into two parts. First we demonstrate confluence for evaluations which terminate normally, i.e., those which yield a value and which apply all of the side effects generated in the course of the expression evaluation. Then we show that if an evaluation sequence exists which leads to undefined behaviour, this undefinedness can not be escaped, and that all states reachable from the initial one must necessarily either be undefined themselves, or still admit the possibility of becoming undefined in one or more steps.

This second result makes it clear that a normal terminating evaluation and an undefined one can not both begin from the same initial state. We reason as follows: assume that such a situation exists. Then our second result states that it is possible to reach undefinedness from the final state of the normal evaluation. But if it is a final state, then it can not take any more steps, and it is not in an undefined state itself because it has yielded a proper value. Thus we have a contradiction and an assurance to the effect that all evaluations are in fact deterministic.

4 Successful evaluations

Even with the above assumption that our reduction sequences do not become undefined, the task of proving determinism for expression evaluation is quite complicated. In particular, the system as described is made difficult to reason about by the fact that side effect applications and other forms of reduction can intermingle. The first stage of our proof is

to demonstrate that side effect applications can all be postponed to the end of an evaluation sequence without affecting the result.

This should be clear from the constraints described earlier: if a side effect application were to make a difference, a subsequent reference to memory would need to look at some part of memory that the side effect had changed; but this is precisely one of those situations forbidden (a reference to updated memory) and would lead to undefinedness, contradicting our earlier assumption.

The proof proceeds by first showing that side effect applications and other reductions can commute.

Lemma 1. *For all expressions e_0, e_1, for all states σ, σ_0, σ_1, and for all side effects η, if η is pending in σ, with $\sigma_0 = \texttt{apply\_se}(\sigma, \eta)$ (i.e., σ_0 is the state that results from applying η to σ), and $\langle e_0, \sigma_0\rangle \rightarrow_e \langle e, \sigma_1\rangle$ then there exists a state σ' such that $\langle e_0, \sigma\rangle \rightarrow_e \langle e, \sigma'\rangle$, η is pending in σ' and $\sigma_1 = \texttt{apply\_se}(\sigma', \eta)$.*

This is a straightforward rule induction on the inductive definition of $\rightarrow_e$. Another induction readily extends this to allow side effect applications to be pushed past any number of other expression reduction steps. Using this, we then induct on the number of reductions to prove our "separation theorem":

Theorem 1 (Separation). *For all expressions e_0, e, and for all states σ_0, σ, if $\langle e_0, \sigma_0\rangle \rightarrow_e^* \langle e, \sigma\rangle$, then there exists a state σ' and a sequence of side effects $\eta_1 \ldots \eta_n$ where both the* update_map*s and memory contents of σ_0 and σ' are the same, and $\langle e_0, \sigma_0\rangle \rightarrow_e^* \langle e, \sigma'\rangle$ and σ is the result of applying the side effects $\eta_1 \ldots \eta_n$ to σ'.*

(Note that the final value e is present after the expression reduction steps, and before the side effect applications begin. This is because these later applications can not change the value that an expression yields.)

We now consider the $\rightarrow_e$ relation as the union of two components: reductions where no side effect applications occur, and reductions that are exclusively side effect applications. Let us use $\rightarrow_E$ for the former and $\rightarrow_A$ for the latter so that $\rightarrow_e = \rightarrow_E \cup \rightarrow_A$. Confluence for both $\rightarrow_E$ and $\rightarrow_A$, together with the separation theorem imply confluence for $\rightarrow_e$ as follows:

1. Consider two reduction sequences starting at $\langle e_0, \sigma_0\rangle$ that both complete normally. One is to $\langle e_1, \sigma_1\rangle$ and the other is to $\langle e_2, \sigma_2\rangle$.

2. By the separation theorem, both reduction sequences can be separated into two phases, with intermediate points $\langle e_1, \sigma_1' \rangle$ and $\langle e_2, \sigma_2' \rangle$, such that $\langle e_0, \sigma_0 \rangle \rightarrow_E^* \langle e_1, \sigma_1' \rangle$ and $\langle e_1, \sigma_1' \rangle \rightarrow_A^* \langle e_1, \sigma_1 \rangle$ (similarly for e_2 etc.)
3. Because e_1 and e_2 represent completed evaluations, they must be values. As $\rightarrow_A$ only applies side effects, it doesn't change expressions. Thus the states reached by $\rightarrow_E^*$ must be terminal with respect to it. Then if $\rightarrow_E$ is confluent, these intermediate states are actually the same.
4. Now we have two reduction sequences involving $\rightarrow_A^*$ from the same starting point. As $\rightarrow_A$ is also confluent, the final states are necessarily identical.

Given this result, we need only prove that $\rightarrow_E$ and $\rightarrow_A$ are confluent.

4.1 Confluence for $\rightarrow_E$

We establish confluence for $\rightarrow_E$ by demonstrating a diamond property for single steps of the relation.

Before beginning a proof such as this, it is instructive to consider parallels with the similar task that one faces in attempting to prove confluence for the λ-calculus. There, things are somewhat complicated by the fact that a reduction in the RHS of a β-redex may have to be matched by many repetitions of essentially the same reduction in an alternative branch where the RHS has been substituted into the body of the LHS. This doesn't happen in the Cholera semantics, where substitution doesn't arise.

However, the λ-calculus is at least entirely syntax-directed; if a redex is present, then the reduction can always take place, and its result will always be the same. Reductions in the λ-calculus can be said to ignore their context. This is not the case in Cholera where the accompanying state, an ever-present and varying context, can affect reductions. This is not just a matter of different values for variables affecting the value of an expression, but more significant: a state with a large `update_map` may make a reduction that would otherwise turn a variable into a value instead produce undefinedness.

With this motivation behind us, the first stage in our proof will be to characterise the degree to which states can vary and yet still produce the same reduction for a given piece of syntax. Furthermore, because

expression reductions affect the state[4], we want to characterise the way in which this happens, so that, ultimately, we will be able to state that reduction x can reduce in the same way both before and after reduction y.

Theorem 2 (Reduction characterisation). *If* $\langle e_0, \sigma_0 \rangle \rightarrow_E \langle e, \sigma \rangle$ *then there exists a function* f *characterising the reduction, such that* $f(\sigma_0) = \sigma$*, and for all* σ_0' *which are "no more restrictive" than* σ_0*, then* $\langle e_0, \sigma_0' \rangle \rightarrow_E \langle e, f(\sigma_0') \rangle$.

The meaning of "no more restrictive" above turns out to be rather detailed in its expression, really suitable only for the consumption of a theorem prover. In essence it requires that the `update_map` be no bigger in σ_0' than it is in σ_0, but there are also are a number of conditions required of both the initial states and the expressions involved. One of these is that e_0 be well-typed. Another is that e not be undefined; computations that do allow e to become undefined are discussed in section 5.

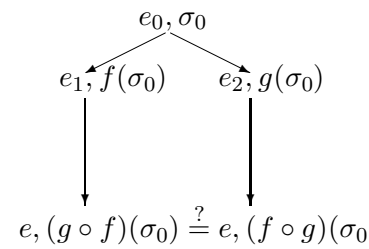
We also have the following important lemma, which like the previous is established by induction over the reduction relation.

Lemma 2 (Reduction preconditions preserved). *If* $\langle e_0, \sigma_0 \rangle \rightarrow_E \langle e, \sigma \rangle$*, then* σ *is no more restrictive than* σ_0 *in the sense of theorem 2.*

Now we can prove the diamond property for $\rightarrow_E$ relatively straightforwardly. Again an induction is required over the reduction relation. The inductive or "sub-expression" cases, where we have two reductions within the same sub-expression, are handled by the inductive hypotheses, so it is just the cases where an expression form admits two reductions in different sub-expressions which prove difficult. This includes both the normal binary operators, and also assignment, which needs to be treated separately because unlike the other operators, it adds a side effect to those pending.

In such a situation, our reduction characterisation and reduction preconditions results tell us immediately that a "diamond" of four sides can be constructed. If the functions required to exist by the first result are f and g, then the diagram looks like:

[4] Though $\rightarrow_E$ holds `update_map`s and thus memory constant, we will still get new side effects being added to the queue of those pending, and as objects are referred to, `ref_map`s also increase.



The question then remains as to whether or not f and g will commute. They do in fact, as each does little more than specify the additions to the starting state's `ref_map` and pending side effects. Addition on bags being commutative, the result follows.

4.2 Confluence for $\rightarrow_A$

The second requirement of the proof of is to show that the $\rightarrow_A$ relation is confluent. We show this by demonstrating a diamond property. This is a considerably simpler task than for $\rightarrow_E$.

Recall that we are performing reductions in a context where all of the side effects can be applied successfully, resulting in normal termination with a value. This implies that no pair of pending side effects affect overlapping parts of memory. We show this by contradiction. One of the side effects must have been applied first. Subsequent to this application, the other side effect can not have been applied because this would result in undefined behaviour (two updates of the same part of memory). But if the second side effect is not applied, then the final state must still have side effects pending, which also contradicts our assumption, because a normal termination is a sequence point, by which state all side effects must have been applied.

So, all of the side effects affect different parts of memory, and can therefore be applied independently of one another. The required diamond property is an immediate consequence of this.

5 Undefined evaluations

We begin by defining *state safety*. A state is *safe* if none of its pending side effects conflict neither with each other (i.e., do not affect overlapping parts of memory), nor with the state's `ref_map` and `update_map`. It should be clear that a state which is safe can apply all of its side effects without becoming undefined. The converse is the basis of our first lemma in this section.

Lemma 3 (Finite and unsafe states can become undefined). *If a state σ_0 is both unsafe and has a finite bag of pending side effects, then for all e_0 there exists a reduction sequence such that $\langle e_0, \sigma_0\rangle \rightarrow_e^* \langle \mathcal{U}\rangle$, where $\mathcal{U}$ represents undefinedness.*

Also, for all e_0, e, σ_0 and σ, if σ_0 is unsafe, and $\langle e_0, \sigma_0\rangle \rightarrow_e \langle e, \sigma\rangle$, then σ is also unsafe.

Cholera represents undefinedness arising as a result of expression evaluation (e.g., division by zero, or a reference to a variable already updated) by replacing the offending expression with $\mathcal{U}$ in the syntax tree and then letting this "bubble" its way to the top of the tree. This can not be prevented.

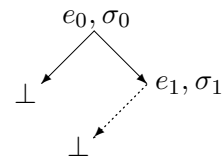
Lemma 4 (Undefined sub-expressions can always ascend). *If an expression e_0 contains $\mathcal{U}$ as a sub-expression, then for all σ_0 there exists a reduction sequence such that $\langle e_0, \sigma_0\rangle \rightarrow_e^* \langle \mathcal{U}\rangle$.*

Also, for all e_0, e, σ_0 and σ, if e_0 has an undefined sub-expression, and $\langle e_0, \sigma_0\rangle \rightarrow_e \langle e, \sigma\rangle$, then e must also have an undefined sub-expression (where e itself may be that undefined sub-expression).

These two results (neither of which is particularly surprising) make it clear that a large class of expression-state pairs, those which are unsafe or which have undefined subexpressions, though not necessarily "fully undefined", might as well be. We shall refer to such states as *effectively undefined*. Though a state's being effectively undefined may not seem such a strong claim initially, the condition preservation clauses of the lemmas above should make it clear that an effectively undefined state is one which can never yield a value. In conjunction with the fact that all sequence point free expressions must terminate[5], we can see that effectively undefined means "will necessarily become undefined".

Our next theorem is more significant. We wish to show that if a reduction occurs which makes something effectively undefined, when it was not effectively undefined before, then if one takes a different step from the same initial state, the result will either be effectively undefined, or it will retain the ability to make a reduction to an effectively undefined state. This can be represented as a "broken" diamond:

[5] Sequence point free expressions do not include function applications.



Another analogy is that of the cliff-edge. Over the edge lies effective undefinedness. Once one reaches the edge, one can walk along it, but while it may be possible to avoid falling over the edge for some indeterminate length of time, it is not possible to move away. The proof proceeds in a similar way to that of the proof of the confluence of $\rightarrow_E$.

While the inductive cases are straightforward, we need to cope with the fact that the reduction from $\langle e_0, \sigma_0 \rangle$ to $\langle e_1, \sigma_1 \rangle$ might involve a reduction in a sub-expression unrelated to that which produced the undefinedness. Inside $\langle e_1, \sigma_1 \rangle$ we want to have a reduction occur that is analogous to the one that produced undefinedness from $\langle e_0, \sigma_0 \rangle$. We do this by again establishing a reduction characterisation result, and by demonstrating that reductions preserve this.

In this case, the characterisation is essentially that an analogous reduction to undefinedness can occur in any state that is at least as restrictive as the original. This condition is preserved both by $\rightarrow_E$ and $\rightarrow_A$.

We then do an induction of the number of steps along the cliff's edge to produce:

Theorem 3 (The cliff's edge). *For all e_0, σ_0, if $\langle e_0, \sigma_0 \rangle \rightarrow_e \langle e_1, \sigma_1 \rangle$ and $\langle e_1, \sigma_1 \rangle$ is effectively undefined, then for all e_2 and σ_2 such that $\langle e_0, \sigma_0 \rangle \rightarrow_e^* \langle e_2, \sigma_2 \rangle$, there exists e' and σ' such that $\langle e_2, \sigma_2 \rangle \rightarrow_e \langle e', \sigma' \rangle$, and $\langle e', \sigma' \rangle$ is effectively undefined.*

We still need to add one more diamond property. This is a surprisingly easy proof as it does not require an induction over the meaning relation. Instead the characterisation functions and our lemma (1) that $\rightarrow_E$ and $\rightarrow_A$ commute combine to give:

Theorem 4 (A diamond property for $\rightarrow_E$ and $\rightarrow_A$). *For all e_0, e_1, e_2 σ_0, σ_1, σ_2: if $\langle e_0, \sigma_0 \rangle \rightarrow_E \langle e_1, \sigma_1 \rangle$ and $\langle e_0, \sigma_0 \rangle \rightarrow_A \langle e_2, \sigma_2 \rangle$ and both $\langle e_1, \sigma_1 \rangle$ and $\langle e_2, \sigma_2 \rangle$ are not effectively undefined, then there exist e and σ (possibly effectively undefined), such that $\langle e_1, \sigma_1 \rangle \rightarrow_A \langle e, \sigma \rangle$ and $\langle e_2, \sigma_2 \rangle \rightarrow_E \langle e, \sigma \rangle$.*

The final proof is now possible. We wish to show that if a reduction sequence takes an initial state ($\langle e_0, \sigma_0 \rangle$) to undefinedness, then all other

possible destinations from the same starting point retain this possibility. In essence, we exploit the possibility of completing a confluent diamond on the cliff-tops.

1. We have a reduction sequence from $\langle e_0, \sigma_0 \rangle$ to undefinedness. Let $\langle e_1, \sigma_1 \rangle$ be the last state in this sequence not effectively undefined.
2. We have another reduction sequence to $\langle e_2, \sigma_2 \rangle$, and by assumption this is not effectively undefined.
3. Therefore, using all three of our diamond properties for $\rightarrow_E$ and $\rightarrow_A$, we have a common possible destination for both $\langle e_1, \sigma_1 \rangle$ and $\langle e_2, \sigma_2 \rangle$. Call this $\langle e, \sigma \rangle$.
4. Having come along the cliff's edge from $\langle e_1, \sigma_1 \rangle$, $\langle e, \sigma \rangle$ must still be on the edge, thereby retaining the possibility of a reduction to an effectively undefined state, if it is not an effectively undefined state already.
5. Effectively undefined states all allow for a reduction sequence to "full" undefined-ness, so $\langle e_2, \sigma_2 \rangle$ must do so as well by virtue of being able to reduce to $\langle e, \sigma \rangle$.

6 Conclusion

The fact that we have ended with proofs of diamond properties for $\rightarrow_E$, $\rightarrow_A$ and $\rightarrow_E$ *vs.* $\rightarrow_A$ may suggest that the rather specialised proof strategy used in section 4.1 might as well have been subsumed into an all-encompassing proof of confluence for the whole meaning relation. In particular, it is easy to see with hindsight that demonstrating a diamond property, where neither reduction is to an effectively undefined destination, would have been reasonably straightforward. Nonetheless, the only theorem that becomes redundant in this alternative proof is the separation result (theorem 1). All the other results given are necessary parts of either proof.

It is extremely important that this work was built on the support provided by mechanical theorem proving (HOL, in this case). It would have been unimaginable without that support. The proof script for proving this result is almost 6000 lines of SML code (excluding comments). This work is thus a demonstration of both the importance and utility of mechanised theorem-proving. The mechanisation of the semantics ensures that one can be sure of one's results, and that no details have been overlooked. In this work, the diamond proofs in question involved analysis of many (approximately 200) different cases corresponding to a pair-wise examination of all the possible ways in which all possible expressions might

evolve. Such a proof done by hand would be inevitably subject to question because of the high possibility of error. With HOL's help, the possibility of error has been eliminated.

This work is also valuable because it demonstrates an interesting result about the programming language C. This in turn is a demonstration that the practical formalisation of programming language semantics is not an impossible dream. In [8], Ritchie says "the C standard did not attempt to specify formally the language semantics, and so there can be dispute over fine points". In the formal setting provided by Cholera, fine points are no longer the subject of dispute: not only does the language gain an unambiguous specification, it is also possible to state the definition's consequences with certainty.

References

1. M. J. C. Gordon and T. Melham. *Introduction to HOL: a theorem proving environment*. Cambridge University Press, 1993.
2. Matthew Hennessy. *The semantics of programming languages*. John Wiley and Sons, 1990.
3. *Programming languages – C*, 1990. ISO/IEC 9899:1990.
4. ISO committee JTC1/SC22/WG14. Record of responses. Available from `ftp://ftp.dmk.com/DMK/sc22wg14/rr/`.
5. Michael Norrish. Derivation of verificiation rules for C from operational definitions. In J. von Wright, J. Grundy, and J. Harrison, editors, *Supplementary proceedings of TPHOLs '96*, number 1 in TUCS General Publications, pages 69–75. Turku Centre for Computer Science, August 1996.
6. Michael Norrish. An abstract dynamic semantics for C. Technical Report 421, Computer Laboratory, University of Cambridge, May 1997.
7. Michael Norrish. *C formalised in HOL*. PhD thesis, Computer Laboratory, University of Cambridge, 1998. Submitted August, 1998.
8. D. M. Ritchie. The development of the C language. *ACM SIGPLAN Notices*, 28(3):201–208, March 1993.

A Programming Logic for Sequential Java

Arnd Poetzsch–Heffter and Peter Müller

Fernuniversität Hagen, D-58084 Hagen, Germany
{Arnd.Poetzsch-Heffter, Peter.Mueller}@Fernuni-Hagen.de

Abstract. A Hoare-style programming logic for the sequential kernel of Java is presented. It handles recursive methods, class and interface types, subtyping, inheritance, dynamic and static binding, aliasing via object references, and encapsulation. The logic is proved sound w.r.t. an SOS semantics by embedding both into higher-order logic.

1 Introduction

Java is a practically important object-oriented programming language. This paper presents a logic to verify sequential Java programs. The motivations for investigating the logical foundations of Java are as follows:

1. Java plays an important role in the quickly developing software component industry and the smart card technology. Verification techniques can be used for static program analysis, e.g., to prove the absence of null-pointer exceptions. The Java subset used in this paper is similar to JavaCard, the Java dialect for implementing smart cards.
2. As pointed out in [MPH97], logical foundations of programming languages form a basis for program specification technology. They allow for expressive specifications (covering e.g., abstraction, sharing-properties, and side-effects) and are needed to assign a formal meaning to interface specifications.
3. Formality is a prerequisite for tool-based verification. Tool support is necessary to keep large program proofs error-free.
4. Java is typical for a large group of OO-languages including C++, Eiffel, Oberon, Modula-3, BETA, and Ada95. The developed techniques can be adapted to other languages of that group.

The goal underlying this research is the development of interactive programming environments that support specification and verification of OO-programs. Some design decisions have been made w.r.t. this goal.

Approach. Three aspects make verification of OO-programs more complex than verification of programs with just recursive procedures and arbitrary pointer data structures: Subtyping, abstract types, and dynamic binding. Subtyping allows variables to hold objects of different types. Abstract types have different or incomplete implementations. Thus, techniques are needed to formulate type

S.D. Swierstra (Ed.): ESOP/ETAPS'99, LNCS 1576, pp. 162–176, 1999.
© Springer-Verlag Berlin Heidelberg 1999

properties without referring to implementations. Dynamic binding destroys the static connection between the calls and the body of a procedure.

Our solutions to these problems build on well-known techniques. Object stores are specified as an abstract data type with operations to create objects and to read and write instance variables/attributes. Based on object stores, abstractions of object structures can be expressed which are used to specify the behavior of abstract types. To express the relation between stores in pre- and poststates, the current object store can be referenced through a special variable.

Our programming logic refines Hoare logics for procedural languages. To handle dynamic binding, the programming logic allows one to prove properties of so-called virtual methods, i.e., methods that capture the common properties of the corresponding methods in subtypes. The distinction between the virtual behavior of a method and the behavior of the associated implementations allows one to transfer verification techniques for procedures to OO-programs.

The logic is proved sound w.r.t. an SOS semantics of the programming language. Since the semantics of modern OO-languages tends to be rather complex, such soundness proofs can become quite long for full-size languages and should therefore be checkable by mechanical proof checkers. To provide a basis for mechanical checking, we embed both semantics into a higher-order logic and derive the axioms and rules of the logic from those of the operational semantics.

Related Work. In [Lei97], a wlp-calculus for an OO-language similar to our Java subset is presented. In contrast to our work, method specifications are part of the programs. The approach in [Lei97] can be considered as restricting our approach to a certain program development strategy (in [PHM98], we discuss this topic). Thereby, it becomes simpler and more appropriate for automatic checking, but gives up flexibility that seems important to us for interactive program development and verification. A different logic for OO-programs that is related to type-systems is presented and proved sound in [AL97]. It is developed for an OO-language in the style of the lambda calculus whereas we are aiming to directly support the verification of an existing practical language. The presented programming logic extends the foundations developed in [PHM98] by covering encapsulation and subclassing. Furthermore, [PHM98] does not discuss the relation between the logic and a formal semantics and does not prove soundness.

In [vON98], type-safety is formally proved for a Java subset similar to ours. Corresponding to our soundness proof, both operational semantics and typing rules are formalized in higher-order logic. However, the type-safety proof has already been mechanically checked in Isabelle. [JvdBH+98] uses an operational semantics of a Java subset to verify various properties of implementations with the PVS proof checker without employing an axiomatic semantics. As will become clear from the presented paper, Hoare-logic provides an additional level of abstraction. This simplifies the handling of subtyping and abstract methods, and proofs become more intuitive. In practice, verification requires elaborate specification techniques like the one described in [Lea96]. In [MPH97], we outline the connection between such specifications and our logic.

Overview. Section 2 presents the operational semantics of the Java kernel, section 3 the programming logic. The soundness proof is contained in Sect. 4.

2 A Semantics for Sequential Java

This section describes the sequential Java kernel, Java-K for short, and presents its dynamic semantics. Compared to Java, Java-K supports only a simple expression and statement syntax, but captures the full complexity of the method invocation semantics. As specification technique we use structural operational semantics (SOS). We assume that the reader is familiar with Java and explain only the restrictions of Java-K. The formal presentation concentrates on those aspects that are needed for the soundness proof in Sect. 4.

Java-K Programs. A Java-K program is a set of type declarations where a type is either a class or an interface. A class declares its name, its superclass, the list of interfaces implemented by the class, and its members. A member is a field, instance method, or static method. Members can be public, protected, or private. Java-K provides the default constructor, but does not support constructor definitions. Method declarations contain the access mode, the method signature, a list of local variables, and a statement as method body. To keep things simple, methods in Java-K have exactly one parameter named p and have always a return type. Overloading is not allowed. The return type, the name of the method, and the parameter of the methods are given by the so-called method signature. An interface declares its name, the list of extended interfaces, and the signatures of its methods:

data type

JavaK-Program	= list of *TypeDecl*
TypeDecl	= *ClassDecl(CTypeId CTypeId ITypeIdList ClassBody)*
	| *InterfaceDecl(ITypeId ITypeIdList InterfaceBody)*
ClassBody	= list of *MemberDecl*
MemberDecl	= *FieldDecl(Mode Type FieldId)*
	| *MethodDecl (Mode MethodSig VarList Statement)*
	| *StaticMethDecl (Mode MethodSig VarList Statement)*
Mode	= *Private()* | *Protected()* | *Public()*
InterfaceBody	= list of *MethodSig*
ITypeIdList	= list of *ITypeId*
MethodSig	= *Sig(Type MethodId Type)*
VarList	= list of *VarDecl*
VarDecl	= *Vardcl(Type VarId)*
Type	= *booleanT()* | *intT()* | *nullT()* | *ct(CTypeId)* | *it(ITypeId)*

Java-K has the predefined types *booleanT*, *intT*, and *nullT* (the type of the null reference), and the user defined class and interface types. The subtype relation on sort *Type* is defined as in Java and denoted by $\preceq$.

An expression in Java-K is an integer or boolean constant, the null reference, a variable or parameter identifier, the identifier "this" (denoting the reference to the object for which the non-static method was invoked), or a unary or binary

expression over relational or arithmetic operators. The statements of Java-K are defined below along with their dynamic semantics.

Capturing Statement Contexts. The semantics of a statement depends on the context of the statement occurrence. We assume that the program context of a statement is always implicitly given and that we can refer to method declarations in this context. Method declarations are denoted by T@m where m is a method name in class T. *MethDeclId* is the sort of such identifiers. The function

$$body: \ MethDeclId \to Statement$$

maps each method declaration to the statement constituting its body. If T is a class type and m a method of T, the function

$$impl: \ Type \times MethodId \to MethDeclId \cup \{undef\}$$

yields the corresponding declaration; otherwise it yields *undef*. Note that T can inherit the declaration of m from a superclass. Similarly to method declaration identifiers, we introduce field declaration identifiers of the form T@a where a is a field name in class T. The sort of such identifiers is denoted by *FieldDeclId*. They are needed to distinguish instance variables with the same field name occurring in one object.

States. A statement is essentially a partial state transformer. A state in Java-K describes (a) the current values for the local variables and for the method parameters p and this, and (b) the current object store. Values in Java-K are either integers, booleans, the null reference, or references to objects of a class type:

data type	$\tau: Value \to Type$	
$Value = b(\ Bool\ )$	$\tau(b(B))$	$= booleanT$
$\mid\ i(\ Int\ )$	$\tau(i(I))$	$= intT$
$\mid\ null()$	$\tau(null)$	$= nullT$
$\mid\ ref(\ CTypeId, ObjId\ )$	$\tau(ref(T, OI))$	$= ct(T)$

Values constructed by *ref* represent the references to objects. The sort *ObjId* denotes some suitable set of object identifiers to distinguish different objects of the same type. The function τ yields the type of a value.

The state of an object is given by the values of its instance variables. We assume a sort *InstVar* for the instance variables of all objects and a function

$$instvar: \ Value \times FieldDeclId \to InstVar \cup \{undef\}$$

where *instvar*(V,T@a) is defined as follows: If V is an object reference and the corresponding object has an instance variable named T@a, this instance variable is returned. Otherwise *instvar* yields *undef*. The state of all objects and the information whether an object is alive (i.e., allocated) in the current program state is formalized by an abstract data type Object Store with sort *Store* and the following functions:

$\_\langle\_ := \_\rangle$	$: Store \times InstVar \times Value$	$\to Store$
$\_\langle\_\rangle$	$: Store \times CTypeId$	$\to Store$
$\_(\_)$	$: Store \times InstVar$	$\to Value$
$alive$	$: Value \times Store$	$\to Bool$
new	$: Store \times CTypeId$	$\to Value$

$OS\langle IV := V\rangle$ yields the object store that is obtained from OS by updating instance variable IV with value V. $OS\langle T\rangle$ yields the object store that is obtained from OS by allocating a new object of type T. $OS(IV)$ yields the value of instance variable IV in store OS. If V is an object reference, $alive(V, OS)$ tests whether the referenced object is alive in OS. $new(OS, TID)$ yields a reference to an object of type $ct(TID)$ that is not alive in OS. Since the properties of these functions are not needed for the soundness proof in Sect. 4, we do not discuss their axiomatization here and refer the reader to [PHM98].

Program states are formalized as mappings from identifiers to values. To have a uniform treatment for variables and the object store, we use \$ as identifier for the current object store:

$$State \equiv (VarId \cup \{\,\mathrm{this},\,\mathrm{p}\,\} \to Value \cup \{undef\}) \times (\{\,\$\,\} \to Store \cup \{undef\})$$

For $S \in State$, we write $S(\mathrm{x})$ for the application to a variable or parameter identifier and $S(\$)$ for the application to the object store. By $S[\mathrm{x} := V]$ and $S[\$:= OS]$ we denote the state that is obtained from S by updating variable x and \$, respectively. The canonical evaluation of expression e in state S is denoted by $\epsilon(S, \mathrm{e})$ yielding an element of sort *Value* or *undef* (note that expressions in Java-K always terminate and do not have side-effects). The state in which all variables are undefined is named *initS*.

Statement Semantics. The semantics of Java-K statements is defined by inductive rules. $S:\ \mathrm{s}\ \to S'$ expresses the fact that executing *Statement* s in *State* S terminates in *State* S'. In the rules, x and y range over variable or parameter identifiers, and e over expressions.

In order to keep the size of the specification manageable, we assume that some Java-K statements are given in a syntax that is decorated with information from type and name analysis. An access to an instance variable a of static type T is written as T@a. Java-K provides statements for reading and writing instance variables with the following semantics (note that the context conditions of Java and the antecedent of the rule guarantee that $instvar(\mathrm{y}, \mathrm{T@a})$ is defined):

$$\frac{S(\mathrm{y}) \neq null}{S:\ \mathrm{x} = \mathrm{y.T@a};\ \to S[\mathrm{x} := S(\$)(instvar(\mathrm{y}, \mathrm{T@a}))]}$$

$$\frac{S(\mathrm{y}) \neq null}{S:\ \mathrm{y.T@a{=}e};\ \to S[\$:= S(\$)\langle instvar(\mathrm{y}, \mathrm{T@a}) := \epsilon(S, \mathrm{e})\rangle]}$$

In Java, there are four kinds of method invocations: (a) invocations of public or protected methods, (b) invocations of private methods, (c) invocations of superclass methods, and (d) invocations of static methods. Invocations of kind (a) are dynamically bound, the others can be bound at compile time. To make the context information visible within the SOS rules, we distinguish kind (a) and (b) syntactically: y.T:m(e) denotes an invocation of kind (a) where T is the static type of y. A statically bound invocation is denoted by y.T@m(e) where T is the class in which m is declared. We can use the same syntax to handle invocations of

superclass methods: A Java method invocation of the form super.m(e) occurring in a class C is in Java-K expressed by a call this.CSuper@m() where CSuper is the nearest superclass of C containing a declaration of m. This way, semantics of invocations of kind (b) and (c) can be given by the same rule. Invocations of kind (d) behave similar, but do not have a this-parameter.

To focus on the interesting aspects, Java-K does not support a return statement. The return value has to be assigned to a local variable "result" that is implicitly declared in all methods. Thus, method invocation means passing the parameters and the object store to the prestate, executing the invoked method, and passing the result and object store back to the invocation environment:

$$\frac{S(\mathrm{y}) \neq \mathit{null},\ \tau(S(\mathrm{y})) \preceq \mathrm{T},\quad \mathit{initS}[\mathrm{this} := S(\mathrm{y}), \mathrm{p} := \epsilon(S,\mathrm{e}), \$:= S(\$)] :\ \mathit{body}(\mathit{impl}(\tau(S(\mathrm{y})),\mathrm{m}))\ \rightarrow S'}{S :\ \mathrm{x{=}y.T{:}m(e)};\ \rightarrow S[\mathrm{x} := S'(\mathrm{result}),\ \$:= S'(\$)]}$$

$$\frac{S(\mathrm{y}) \neq \mathit{null},\ \mathit{initS}[\mathrm{this} := S(\mathrm{y}), \mathrm{p} := \epsilon(S,\mathrm{e}), \$:= S(\$)] :\ \mathit{body}(\mathrm{T@m})\ \rightarrow S'}{S :\ \mathrm{x{=}y.T@m(e)};\ \rightarrow S[\mathrm{x} := S'(\mathrm{result}),\ \$:= S'(\$)]}$$

The rule for the invocation of static methods is identical to the last rule, except that no this-parameter has to be passed. Besides the statements described above, Java-K provides if and while statements, assignment statements with cast, sequential statement composition, and constructor calls. The rules for these statements are straightforward and given in the appendix.

3 A Programming Logic for Java

This section presents a Hoare-style programming logic for Java-K. The logic allows one to formally verify that implementations satisfy interface specifications. For OO-languages, interface specifications are usually given by pre- and postconditions for methods, class invariants, history constraints, etc (cf. e.g. [Lea96]). The formal meaning of such specifications is defined in terms of proof obligations for methods (cf. [PH97]). In this paper, we concentrate on the verification of dynamic properties. For proving properties about the object store, we refer to [PH97]. This section defines the precise syntax of our Hoare triples and explains the axioms and rules of the programming logic.

Specifying Methods and Statements. Properties of methods and statements are expressed by triples of the form $\{\mathbf{P}\}\,\mathrm{comp}\,\{\mathbf{Q}\}$ where $\mathbf{P}$, $\mathbf{Q}$ are sorted first-order formulas and comp is either a statement occurrence within a given Java-K program, a method implementation represented by the corresponding method declaration identifier, or a so-called *virtual method*. Before we clarify the signature over which $\mathbf{P}$, $\mathbf{Q}$ are built, we explain the concept of virtual methods.

Virtual Methods. Java-K supports dynamically bound method invocations. E.g., if T is an interface type, and T1, T2 are classes implementing T, an invocation y.T:m(e) can lead to the execution of T1@m or T2@m depending of the object

held by y. To verify dynamically bound method invocations, we need method specifications reflecting the properties of all implementations that might be executed. Such specifications express the behavior of the so-called *virtual methods*. For every non-private instance method m declared in or inherited by a type T, there is a virtual method denoted by T:m. (This notation corresponds to the syntax used for the invocation semantics in Sect. 2.) For private and static methods, virtual methods are not needed, because statically bound invocations can be directly handled using the properties of the corresponding method bodies.

Signatures of Pre- and Postconditions. In program specifications, we have to refer to types, fields, and variables in pre- and postconditions. We enable that by introducing constant symbols for these entities. For a given Java-K program, Σ denotes the signature of sorts, functions, and constant symbols as described in Sect. 2. In particular, it contains constant symbols for the types and fields. Furthermore, we treat parameters, program variables, and the variable \$ for the current object store syntactically as constant symbols of sort *Value* and *Store* to simplify quantification and substitution rules and to define context conditions for pre- and postconditions.

A triple { **P** } comp { **Q** } is called a *statement annotation*, *implementation annotation*, or *method annotation* if the syntactical component comp is a statement, method implementation, or virtual method, respectively. Pre- and postconditions of statement annotations are formulas over $\Sigma \cup \{\text{this}, \text{p}, \$\} \cup VAR(\text{m})$ where m is the method enclosing the statement and $VAR(\text{m})$ denotes the set of local variables of m. Preconditions in method annotations or implementation annotations are formulas over $\Sigma \cup \{\text{this}, \text{p}, \$\}$. Postconditions in such annotations are formulas over $\Sigma \cup \{\text{result}, \$\}$.

To handle recursive methods, we use *sequents* of the form $\mathcal{A} \rhd \mathbf{A}$ where $\mathcal{A}$ is a set of method and implementation annotations and **A** is a triple. Triples in $\mathcal{A}$ are called *assumptions* of the sequent and **A** is called the *consequent* of the sequent. Intuitively, a sequent expresses the fact that we can prove a triple based on some assumptions about methods.

Axiomatic Semantics. The axiomatic semantics of Java consists of axioms and rules for statements and methods. The new axioms and rules are described in the following two paragraphs. The standard Hoare rules (e.g., while rule) are presented in Fig. 1. A more detailed discussion of programming logics for OO-languages and their applications is given in [PHM98].

Statements. The cast-axiom is very similar to Hoare's classical assignment axiom. However, to prevent runtime errors, a stronger precondition assures that the type conversion is legal. The constructor-axiom works like an assignment axiom: The new object is substituted for the left-hand-side variable and the modified object store for the initial store. Reading a field substitutes the value held by the addressed instance variable for the left-hand-side variable. Writing field access replaces the initial object store by the updated store:

cast-axiom: $\vdash \{\ \tau(\mathrm{e}) \preceq \mathrm{T} \wedge \mathbf{P}[\mathrm{e/x}]\ \}\ \ \mathrm{x} = (\mathrm{T})\ \mathrm{e};\ \ \{\ \mathbf{P}\ \}$

constructor-axiom: $\vdash \{\ \mathbf{P}[\mathit{new}(\$, \mathrm{T})/\mathrm{x}\ ,\ \$\langle \mathrm{T} \rangle/\$]\ \}\ \mathrm{x} = \text{new T}();\ \{\ \mathbf{P}\ \}$

field-read-axiom: $\vdash \{\ \mathrm{y} \neq \mathit{null} \wedge \mathbf{P}[\$(\mathit{instvar}(\mathrm{y}, \mathrm{S@a}))/\mathrm{x}]\ \}\ \ \mathrm{x} = \mathrm{y.S@a};\ \ \{\ \mathbf{P}\ \}$

field-write-axiom: $\vdash \{\ \mathrm{y} \neq \mathit{null} \wedge \mathbf{P}[\$\langle \mathit{instvar}(\mathrm{y}, \mathrm{S@a}) := \mathrm{e} \rangle/\$]\ \}\ \ \mathrm{y.S@a} = \mathrm{e};\ \ \{\ \mathbf{P}\ \}$

The invocation-rule uses properties of virtual methods to verify invocations of dynamically bound methods. The fact that local variables different from the left-hand-side variable are not modified by an invocation is expressed by the *invocation-var-rule* that allows one to substitute logical variables Z in pre- and postconditions by local variables w (w different from x):

invocation-rule:
$$\frac{\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ \ \mathrm{T{:}m}\ \ \{\ \mathbf{Q}\ \}}{\mathcal{A} \vdash \{\ \mathrm{y} \neq \mathit{null} \wedge \mathbf{P}[\mathrm{y/this, e/p}]\ \}\ \ \mathrm{x} = \mathrm{y.T{:}m(e)};\ \{\ \mathbf{Q}[\mathrm{x/result}]\ \}}$$

invocation-var-rule:
$$\frac{\mathcal{A} \vdash \{\ \mathbf{P}\ \} \qquad \mathrm{x} = \mathrm{y.T{:}m(e)};\ \ \{\ \mathbf{Q}\ \}}{\mathcal{A} \vdash \{\ \mathbf{P}[\mathrm{w}/Z]\ \}\ \ \mathrm{x} = \mathrm{y.T{:}m(e)};\ \ \{\ \mathbf{Q}[\mathrm{w}/Z]\ \}}$$

Static methods are bound statically. Therefore, method implementations are used instead of virtual methods to verify invocations. In a similar way, method implementations are used to verify calls of private methods and invocations using super. In both cases, the implementation to be executed can be determined statically. The var-rules for static invocations and calls can be found in Fig. 1.

static-invoc-rule:
$$\frac{\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ \ \mathrm{T@m}\ \ \{\ \mathbf{Q}\ \}}{\mathcal{A} \vdash \{\ \mathbf{P}[\mathrm{e/p}]\ \}\ \ \mathrm{x} = \mathrm{T.m(e)};\ \{\ \mathbf{Q}[\mathrm{x/result}]\ \}}$$

call-rule:
$$\frac{\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ \ \mathrm{T@m}\ \{\ \mathbf{Q}\ \}}{\mathcal{A} \vdash \{\ \mathrm{y} \neq \mathit{null} \wedge \mathbf{P}[\mathrm{y/this, e/p}]\ \}\ \ \mathrm{x} = \mathrm{y.T@m(e)};\ \{\ \mathbf{Q}[\mathrm{x/result}]\ \}}$$

Methods. This paragraph presents the rules to prove properties of method implementations and virtual methods. Essentially, an annotation of a method implementation m holds if it holds for its body. In order to handle recursion, the method annotation may be assumed for the proof of the body. Informally, this is sound, because in any terminating execution, the last incarnation does not contain a recursive invocation of the method:

implementation-rule:
$$\frac{\mathcal{A}\ ,\ \{\mathbf{P}\}\ \ \mathrm{T@m}\ \ \{\mathbf{Q}\} \vdash \{\mathrm{this} \neq \mathit{null} \wedge \mathbf{P}\}\ \ \mathit{body}(\mathrm{T@m})\ \ \{\mathbf{Q}\}}{\mathcal{A} \vdash \{\mathbf{P}\}\ \ \mathrm{T@m}\ \ \{\mathbf{Q}\}}$$

Virtual methods have been introduced to model dynamically bound methods. I.e., a method annotation for T:m reflects the common properties of all implementations that might be executed on invocation of T:m. If T is a class, there are two obligations to prove an annotation $\mathbf{A}$ of a virtual method T:m : 1. Show that the corresponding implementation satisfies $\mathbf{A}$ if invoked for objects of type T. 2. Show that $\mathbf{A}$ holds for objects of proper subtypes of T. The second obligation and annotations of interface type methods can be proved by the *subtype-rule*: If S is a subtype of T, an invocation of T:m on an S object is equivalent to an invocation of S:m. Thus, all properties of S:m carry over to T:m as long as T:m is applied to objects of type S:

class-rule:

$$\frac{\mathcal{A} \rhd \{\ \tau(\text{this}) = \text{T} \wedge \mathbf{P}\ \}\ \mathit{impl}(\text{T,m})\ \{\ \mathbf{Q}\ \} \qquad \mathcal{A} \rhd \{\ \tau(\text{this}) \prec \text{T} \wedge \mathbf{P}\ \}\ \text{T:m}\ \{\ \mathbf{Q}\ \}}{\mathcal{A} \rhd \{\ \tau(\text{this}) \preceq \text{T} \wedge \mathbf{P}\ \}\ \text{T:m}\ \{\ \mathbf{Q}\ \}}$$

subtype-rule:

$$\frac{\text{S} \preceq \text{T} \qquad \mathcal{A} \rhd \{\ \tau(\text{this}) \preceq \text{S} \wedge \mathbf{P}\ \}\ \text{S:m}\ \{\ \mathbf{Q}\ \}}{\mathcal{A} \rhd \{\ \tau(\text{this}) \preceq \text{S} \wedge \mathbf{P}\ \}\ \text{T:m}\ \{\ \mathbf{Q}\ \}}$$

The subtype-rule enables one to prove an annotation for a particular subtype S. To prove the sequent $\mathcal{A} \rhd \{\ \tau(\text{this}) \prec \text{T} \wedge \mathbf{P}\ \}\text{T:m}\{\ \mathbf{Q}\ \}$ let us first assume that the given program is not open to further extensions, i.e., all subtypes $\text{S}_1, \ldots, \text{S}_k$ of T are known. Based on a complete axiomatization of the (finite) subtype relation, we can derive $S_0 \prec \text{T} \Leftrightarrow S_0 \preceq \text{S}_1 \vee \ldots \vee S_0 \preceq \text{S}_k$. Thus, we can prove the sequent by applying the subtype-rule for all subtypes of T and by using the disjunct-rule (see Fig. 1) and strengthening with the above equivalence.

Usually, object-oriented programs are open to extensions; i.e., they are designed to be used as parts of bigger programs containing additional subtypes. Typical examples of such open programs are libraries. Intuitively, open OO-programs are more difficult to verify because extensions can influence the behavior of virtual methods. To handle open programs, the proof obligations for later added subtypes are collected. When a subtype is added, the corresponding obligations have to be shown. A detailed discussion of this topic and a technique how such obligations can be treated as assumptions are given in [PHM98].

Language-Independent Axioms and Rules. Besides the axiomatic semantics, the programming logic for Java contains language-independent axioms and rules to handle assumptions and to establish a connection between the predicate logic of pre- and postconditions and triples of the programming logic (cf. Fig. 1).

4 Towards Formal Soundness Proofs for Complex Programming Logics

The last sections presented two definitions of the semantics of Java-K. The advantage of the operational semantics is that its rules can be used to generate interpreters for validating and testing the language definition (cf. [BCD$^+$89]). The axiomatic definition can be considered as a higher-level semantics and is better suited for verification of program properties. Its soundness should be proved w.r.t. the operational semantics.

Since such soundness proofs can be quite long for full-size programming languages, it is desirable to enable mechanical proof checking (cf. [vON98] for the corresponding argumentation about type safety proofs). That is why we built on the techniques developed by Gordon in [Gor89]: Both semantics are embedded into a higher-order logic in which the axioms and rules of the axiomatic semantics are derived from those of the operational semantics. The application of Gordon's technique to Java-K made extensions necessary: a systematic treatment of SOS rules, and handling of virtual methods and recursion.

while-rule:

$$\frac{\mathcal{A} \rhd \{\ \mathrm{e} = b(\mathit{true}) \wedge \mathbf{P}\ \}\ \mathrm{stm}\ \{\ \mathbf{P}\ \}}{\mathcal{A} \rhd \{\ \mathbf{P}\ \}\ \mathrm{while\ (e)}\ \{\ \mathrm{stm}\ \}\ \ \{\ \mathrm{e} = b(\mathit{false}) \wedge \mathbf{P}\ \}}$$

if-rule:

$$\frac{\mathcal{A} \rhd \{\ \mathrm{e} = b(\mathit{true}) \wedge \mathbf{P}\ \}\ \mathrm{stm1}\ \{\ \mathbf{Q}\ \} \quad \mathcal{A} \rhd \{\ \mathrm{e} = b(\mathit{false}) \wedge \mathbf{P}\ \}\ \mathrm{stm2}\ \{\ \mathbf{Q}\ \}}{\mathcal{A} \rhd \{\ \mathbf{P}\ \}\ \mathrm{if\ (e)}\ \{\ \mathrm{stm1}\ \}\ \mathrm{else}\ \{\ \mathrm{stm2}\ \}\ \ \{\ \mathbf{Q}\ \}}$$

seq-rule:

$$\frac{\mathcal{A} \rhd \{\ \mathbf{P}\ \}\ \mathrm{stm1}\ \{\ \mathbf{Q}\ \} \quad \mathcal{A} \rhd \{\ \mathbf{Q}\ \}\ \mathrm{stm2}\ \{\ \mathbf{R}\ \}}{\mathcal{A} \rhd \{\ \mathbf{P}\ \}\ \mathrm{stm1\ stm2}\ \{\ \mathbf{R}\ \}}$$

call-var-rule:

$$\frac{\mathcal{A} \rhd \{\ \mathbf{P}\ \}\qquad \mathrm{x{=}y.T@m(e);}\ \{\ \mathbf{Q}\ \}}{\mathcal{A} \rhd \{\ \mathbf{P}[\mathrm{w}/Z]\ \}\ \ \mathrm{x{=}y.T@m(e);}\ \{\ \mathbf{Q}[\mathrm{w}/Z]\ \}}$$

where x and w are distinct program variables and Z is an arbitrary logical variable.

static-invoc-var-rule:

$$\frac{\mathcal{A} \rhd \{\ \mathbf{P}\ \}\qquad \mathrm{x{=}T.m(e);}\ \{\ \mathbf{Q}\ \}}{\mathcal{A} \rhd \{\ \mathbf{P}[\mathrm{w}/Z]\ \}\ \ \mathrm{x{=}T.m(e);}\ \{\ \mathbf{Q}[\mathrm{w}/Z]\ \}}$$

where x and w are distinct program variables and Z is an arbitrary logical variable.

false-axiom:

$$\rhd \{\ \mathrm{FALSE}\ \}\ \mathrm{comp}\ \{\ \mathrm{FALSE}\ \}$$

assumpt-axiom:

$$\mathbf{A} \rhd \mathbf{A}$$

assumpt-intro-rule:

$$\frac{\mathcal{A} \rhd \mathbf{A}}{\mathbf{A}_0\ , \mathcal{A} \rhd \mathbf{A}}$$

assumpt-elim-rule:

$$\frac{\mathcal{A} \rhd \mathbf{A}_0 \quad \mathbf{A}_0\ , \mathcal{A} \rhd \mathbf{A}}{\mathcal{A} \rhd \mathbf{A}}$$

conjunct-rule:

$$\frac{\mathcal{A} \rhd \{\ \mathbf{P}_1\ \}\ \mathrm{comp}\ \{\ \mathbf{Q}_1\ \} \quad \mathcal{A} \rhd \{\ \mathbf{P}_2\ \}\ \mathrm{comp}\ \{\ \mathbf{Q}_2\ \}}{\mathcal{A} \rhd \{\ \mathbf{P}_1 \wedge \mathbf{P}_2\ \}\ \mathrm{comp}\ \{\ \mathbf{Q}_1 \wedge \mathbf{Q}_2\ \}}$$

disjunct-rule:

$$\frac{\mathcal{A} \rhd \{\ \mathbf{P}_1\ \}\ \mathrm{comp}\ \{\ \mathbf{Q}_1\ \} \quad \mathcal{A} \rhd \{\ \mathbf{P}_2\ \}\ \mathrm{comp}\ \{\ \mathbf{Q}_2\ \}}{\mathcal{A} \rhd \{\ \mathbf{P}_1 \vee \mathbf{P}_2\ \}\ \mathrm{comp}\ \{\ \mathbf{Q}_1 \vee \mathbf{Q}_2\ \}}$$

strength-rule:

$$\frac{\mathbf{P}' \Rightarrow \mathbf{P} \quad \mathcal{A} \rhd \{\ \mathbf{P}\ \}\ \mathrm{comp}\ \{\ \mathbf{Q}\ \}}{\mathcal{A} \rhd \{\ \mathbf{P}'\ \}\ \mathrm{comp}\ \{\ \mathbf{Q}\ \}}$$

weak-rule:

$$\frac{\mathcal{A} \rhd \{\ \mathbf{P}\ \}\ \mathrm{comp}\ \{\ \mathbf{Q}\ \} \quad \mathbf{Q} \Rightarrow \mathbf{Q}'}{\mathcal{A} \rhd \{\ \mathbf{P}\ \}\ \mathrm{comp}\ \{\ \mathbf{Q}'\ \}}$$

inv-rule:

$$\frac{\mathcal{A} \rhd \{\ \mathbf{P}\ \}\ \mathrm{comp}\ \{\ \mathbf{Q}\ \}}{\mathcal{A} \rhd \{\ \mathbf{P} \wedge \mathbf{R}\ \}\ \mathrm{comp}\ \{\ \mathbf{Q} \wedge \mathbf{R}\ \}}$$

where $\mathbf{R}$ is a Σ-formula, i.e. doesn't contain program variables or \$.

subst-rule:

$$\frac{\mathcal{A} \rhd \{\ \mathbf{P}\ \}\ \mathrm{comp}\ \{\ \mathbf{Q}\ \}}{\mathcal{A} \rhd \{\ \mathbf{P}[t/Z]\ \}\ \mathrm{comp}\ \{\ \mathbf{Q}[t/Z]\ \}}$$

where Z is an arbitrary logical variable and t a Σ-term.

all-rule:

$$\frac{\mathcal{A} \rhd \{\ \mathbf{P}[Y/Z]\ \}\ \mathrm{comp}\ \{\ \mathbf{Q}\ \}}{\mathcal{A} \rhd \{\ \mathbf{P}[Y/Z]\ \}\ \mathrm{comp}\ \{\ \forall Z : \mathbf{Q}\ \}}$$

where Z, Y are arbitrary, but distinct logical variables.

ex-rule:

$$\frac{\mathcal{A} \rhd \{\ \mathbf{P}\ \}\ \mathrm{comp}\ \{\ \mathbf{Q}[Y/Z]\ \}}{\mathcal{A} \rhd \{\ \exists Z : \mathbf{P}\ \}\ \mathrm{comp}\ \{\ \mathbf{Q}[Y/Z]\ \}}$$

where Z, Y are arbitrary, but distinct logical variables.

Fig. 1. Additional axioms and rules

This section outlines the translation of SOS rules into higher-order formulas; it embeds the programming logic of Java-K into higher-order logic and relates it to the operational semantics. Furthermore, it presents the soundness proof for the most interesting rules of the programming logic.

SOS rules in HOL. The SOS rules can be directly translated into a recursive predicate definition of the form:

$$sem(S_i, \mathrm{stm}, S_t) \;\Leftrightarrow_{def}\; \bigvee_{R \in SOS\text{-}rules} (\mathrm{stm} \text{ matches } stmpattern(R) \;\wedge\; antecedents(R)\;)$$

where $stmpattern(R)$ is the statement pattern occurring in the succedent of R and $antecedents(R)$ denotes the antecedents of the rule where free occurrences of logical variables are existentially bound. E.g., the SOS rule for virtual method invocation is transformed to:

$$\begin{aligned}&\mathrm{stm} \text{ matches } (\mathrm{x} = \mathrm{y.T{:}m(e)};) \;\wedge\; \exists S' :\; S_i(\mathrm{y}) \neq null \;\wedge\; \tau(S_i(\mathrm{y})) \preceq \mathrm{T}\\ &\quad \wedge\; sem(initS[\mathrm{this} := S_i(\mathrm{y}), \mathrm{p} := \epsilon(S_i, \mathrm{e}), \$:= S_i(\$)],\; body(impl(\tau(S_i(\mathrm{y})), \mathrm{m})),\; S')\\ &\quad \wedge\; S_t = S_i[\mathrm{x} := S'(\mathrm{result}), \$:= S'(\$)]\end{aligned}$$

The semantics of Java-K is given by the least fixpoint of the defining equivalence for *sem*. To simplify inductive proofs and the embedding of sequents into the semantics framework, we introduce an auxiliary semantics predicate *nsem* with an additional parameter of sort *Nat*:

$$nsem(N, S_i, \mathrm{stm}, S_t) \;\Leftrightarrow_{def}\; \bigvee_{R \in SOS\text{-}rules} (\mathrm{stm} \text{ matches } stmpattern(R) \;\wedge\; antecedents_{nsem}(R)\;)$$

where $antecedents_{nsem}(R)$ is obtained from $antecedents(R)$ by substituting all occurrences of $sem(S, \mathrm{stm}, S')$ by $N > 0 \wedge nsem(N-1, S, \mathrm{stm}, S')$. It is easy to show that *nsem* is monotonous w.r.t. N, i.e., $nsem(N, S_i, \mathrm{stm}, S_t) \Rightarrow nsem(N+1, S_i, \mathrm{stm}, S_t)$. The following lemma relates *sem* and *nsem*:

$$sem(S_i, \mathrm{stm}, S_t) \;\Leftrightarrow\; \exists N : nsem(N, S_i, \mathrm{stm}, S_t)$$

Semantics for Triples and Sequents. To embed triples into HOL, we consider the pre- and postconditions as predicates on states, i.e., as functions from *State* to *Boolean*. A triple of the form $\{\ \mathbf{P}\ \}\ \mathrm{comp}\ \{\ \mathbf{Q}\ \}$ is viewed as an abbreviation for $H(\lambda S.\mathbf{P}^*,\ \mathrm{comp}\ , \lambda S.\mathbf{Q}^*)$ where $\mathbf{P}^*$ and $\mathbf{Q}^*$ are obtained from $\mathbf{P}$ and $\mathbf{Q}$ by substituting all occurrences of program variables v, parameters p, and the constant symbol \$ by $S(\mathrm{v})$, $S(\mathrm{p})$, and $S(\$)$ (for simplicity, we assume here that S does not occur in $\mathbf{P}$ or $\mathbf{Q}$). Based on this syntactical embedding, we can define the semantics of triples in terms of *sem*:

$$\begin{aligned}H(P, \mathrm{stm}, Q) \quad &\Leftrightarrow\; \forall S, S' : P(S) \wedge sem(S, \mathrm{stm}, S') \Rightarrow Q(S')\\ H(P, \mathrm{T@m}, Q) \;&\Leftrightarrow\; H(\ \lambda S.\ S(\mathrm{this}) \neq null \wedge P(S),\ body(\mathrm{T@m}),\ Q)\\ H(P, \mathrm{T_0{:}m}, Q) \;&\Leftrightarrow\; \textstyle\bigwedge_{class(\mathrm{T}), \mathrm{T} \preceq \mathrm{T_0}} H(\ \lambda S.\ \tau(S(\mathrm{this})) = \mathrm{T} \wedge P(S),\ impl(\mathrm{T}, \mathrm{m}),\ Q)\end{aligned}$$

The first equivalence formulates the usual meaning of Hoare triples for statements (cf. [Gor89]). The second defines implementation annotations in terms of the method body. The third expresses the concept of virtual methods: A virtual method abstracts the properties of all corresponding implementations. The conjunct ranges over all class types T that are subtypes of $\mathrm{T_0}$.

The most interesting aspect of the embedding is the treatment of sequents and rules. Sequents cannot be directly translated into implications with assumptions as premises and consequents as conclusions. For the implementation-rule, this translation would lead to the following **incorrect** rule:

$$\frac{\mathcal{A} \wedge H(P, \mathrm{T@m}, Q) \;\Rightarrow\; H(\,\lambda S.\, S(\mathrm{this}) \neq \mathit{null} \wedge P(S),\; \mathit{body}(\mathrm{T@m}),\; Q)}{\mathcal{A} \;\Rightarrow\; H(P, \mathrm{T@m}, Q)}$$

Using the second equivalence, we can show that the antecedent is a tautology. Since $\mathcal{A}$ can be empty, the rule would allow one to prove that implementations satisfy arbitrary properties. The implementation-rule implicitly contains an inductive argument that has to be made explicit in the embedding. This is done using a predicate K that is related to *nsem* just as H is related to *sem*:

$$\begin{array}{ll}
K(N, P, \mathrm{stm}, Q) & \Leftrightarrow\; \forall S, S' : P(S) \wedge \mathit{nsem}(N, S, \mathrm{stm}, S') \Rightarrow Q(S') \\
K(0, P, \mathrm{T@m}, Q) & \Leftrightarrow\; \mathit{true} \\
K(N+1, P, \mathrm{T@m}, Q) & \Leftrightarrow\; K(N,\; \lambda S.\, S(\mathrm{this}) \neq \mathit{null} \wedge P(S),\; \mathit{body}(\mathrm{T@m}),\; Q) \\
K(N, P, \mathrm{T_0{:}m}, Q) & \Leftrightarrow \bigwedge_{\substack{\mathit{class}(\mathrm{T}), \\ \mathrm{T} \preceq \mathrm{T_0}}} K(N,\; \lambda S.\tau(S(\mathrm{this})) = \mathrm{T} \wedge P(S),\; \mathit{impl}(\mathrm{T}, \mathrm{m}),\; Q)
\end{array}$$

Using the lemma that relates *sem* and *nsem*, it is easy to show that

$$H(P,\ \mathrm{comp}\ , Q) \;\Leftrightarrow\; \forall N : K(N, P,\ \mathrm{comp}\ , Q)$$

Based on K, sequents can be directly embedded into HOL. A sequent of the form $\{\mathbf{P}_1\}\mathrm{m}_1\{\mathbf{Q}_1\}$,...,$\{\mathbf{P}_l\}\mathrm{m}_l\{\mathbf{Q}_l\} \rhd \{\mathbf{P}\}$ comp $\{\mathbf{Q}\}$ is considered as abbreviation for

$$\forall N : (\; K(N, P_1, \mathrm{m}_1, Q_1) \wedge \ldots \wedge K(N, P_l, \mathrm{m}_l, Q_l) \Rightarrow K(N, P,\ \mathrm{comp}\ , Q)\;)$$

Because of the relation between H and K, a sequent without assumptions is equivalent to the semantics of triples described by H. The complexity of the embedding is a strong argument for using Hoare rules in practical verification instead of the axiomatization of the operational semantics. Many of the proof steps encapsulated in the soundness proof have to be done again and again when verification is directly based on the rules of the operational semantics.

Soundness of the Programming Logic. In the last paragraph, we formalized the semantics of triples and sequents in terms of *sem* and *nsem*. Having a semantics for the sequents, we can prove the soundness of the Java-K logic. The embedding into HOL was chosen in such a way that the soundness proof can be done separately for each logical rule. We illustrate the needed proof techniques by showing the soundness of the implementation- and the invocation-rule. In the proofs, we abbreviate $S(\mathrm{x}) \neq \mathit{null}$ by $\nu(\mathrm{x})$.

implementation-rule. The soundness proof of the implementation-rule illustrates the implicit inductive argument of that rule and demonstrates the treatment of assumptions. We show:

$$\begin{array}{l}
\big(\forall M : A(M) \;\wedge\; K(M, P, \mathrm{T@m}, Q) \Rightarrow K(M, \lambda S.\nu(\mathrm{this}) \wedge P(S), \mathit{body}(\mathrm{T@m}), Q)\big) \\
\Rightarrow\; \forall N : A(N) \;\Rightarrow\; K(N, P, \mathrm{T@m}, Q)
\end{array}$$

where $A(L)$ denotes the conjunction of the embedded assumptions and P and Q abbreviate $\lambda S.\mathbf{P}^*$ and $\lambda S.\mathbf{Q}^*$, respectively. The proof runs by induction on N:

Induction base for $N = 0$: $K(0, P, \mathrm{T@m}, Q)$ is true by definition of K.
Induction step: Assuming that the hypothesis holds for N.

$\big(\forall M : A(M) \wedge K(M, P, \mathrm{T@m}, Q) \Rightarrow K(M, \lambda S.\nu(\mathrm{this}) \wedge P(S), \mathit{body}(\mathrm{T@m}), Q)\big)$
$\Rightarrow$ [Conjoining the induction hypothesis]
$\big(\forall M : A(M) \wedge K(M, P, \mathrm{T@m}, Q) \Rightarrow K(M, \lambda S.\nu(\mathrm{this}) \wedge P(S), \mathit{body}(\mathrm{T@m}), Q)\big)$
$\wedge \big(A(N) \Rightarrow K(N, P, \mathrm{T@m}, Q)\big)$
$\Rightarrow$ [Instantiate M by N & propositional logic]
$A(N) \Rightarrow K(N, \lambda S.\nu(\mathrm{this}) \wedge P(S), \mathit{body}(\mathrm{T@m}), Q)$
$\Rightarrow$ [Definition of K]
$A(N) \Rightarrow K(N + 1, P, \mathrm{T@m}, Q)$
$\Rightarrow$ [$A(N + 1) \Rightarrow A(N)$, see below]
$A(N + 1) \Rightarrow K(N + 1, P, \mathrm{T@m}, Q)$

The implication $A(N + 1) \Rightarrow A(N)$ follows from the definition of K and the monotonicity of *nsem*.

invocation-rule. The soundness proof of the invocation-rule demonstrates how substitution is handled and why the restrictions on the signatures of pre- and postcondition formulas are necessary. We simplify the proof a bit by leaving out the assumptions in the antecedent and succedent of the rule. The extension to the complete proof is straightforward. Thus, we have to show:

$\forall M : K(M, \lambda S.\mathbf{P}^*, \mathrm{T{:}m}, \lambda S.\mathbf{Q}^*)$
$\Rightarrow \forall N : K(N, \lambda S.\nu(\mathrm{y}) \wedge (\mathbf{P}[\mathrm{y/this, e/p}])^*, \mathrm{x{=}y.T{:}m(e)_;}, \lambda S.(\mathbf{Q}[\mathrm{x/result}])^*)$

Assuming the premise, we prove the conclusion, i.e., for arbitrary N, S, S'':

$\nu(\mathrm{y}) \wedge (\mathbf{P}[\mathrm{y/this, e/p}])^* \wedge \mathit{nsem}(N, S, \mathrm{x{=}y.T{:}m(e)_;}, S'') \Rightarrow (\lambda S.(\mathbf{Q}[\mathrm{x/result}])^*)(S'')$

This is proved by case distinction on N:
Case $N = 0$: From the definition of *nsem* we get that the premise is false.
Case $N > 0$: The following lemma relates substitution and state update:

$(\mathbf{P}[t_1/x_1, \ldots, t_n/x_n])^* = (\lambda S.\mathbf{P}^*)(S[x_1 := \epsilon(S, t_1), \ldots, x_n := \epsilon(S, t_n)])$

By this lemma and with P for $\lambda S.\mathbf{P}^*$ and Q for $\lambda S.\mathbf{Q}^*$, the proof goal becomes:

$\nu(\mathrm{y}) \wedge P(S[\mathrm{this} := S(\mathrm{y}), \mathrm{p} := \epsilon(S, \mathrm{e})]) \wedge \mathit{nsem}(N, S, \mathrm{x{=}y.T{:}m(e)_;}, S'')$
$\Rightarrow Q(S''[\mathrm{result} := S''(\mathrm{x})])$

To show this, we will use the following implication (+):

$\nu(\mathrm{y}) \wedge P(\sigma) \wedge \tau(S(\mathrm{y})) \preceq \mathrm{T} \wedge \mathit{nsem}(N - 1, \sigma, \mathit{body}(\mathit{impl}(\tau(S(\mathrm{y})), \mathrm{m})), S') \Rightarrow Q(S')$

where σ abbreviates $\mathit{initS}[\mathrm{this} := S(\mathrm{y}), \mathrm{p} := \epsilon(S, \mathrm{e}), \$:= S(\$)]$. The proof of (+) uses the general proof assumption $\forall M : K(M, P, \mathrm{T{:}m}, Q)$, the definition of K, and the fact that $\tau(S(\mathrm{y}))$ is a class type. $\tau(S(\mathrm{y}))$ is a class type, because the context conditions of Java/Java-K imply that y is of a reference type and because Java and thus Java-K are type-safe. In addition to (+), we need the fact that for any state S_0 the value of $P(S_0)$ only depends on $S_0(\mathrm{this})$, $S_0(\mathrm{p})$, and $S_0(\$)$, because other variables are not allowed within P (cf. Sect. 3), i.e.,

$S_0(\mathrm{this}) = S_1(\mathrm{this}) \wedge S_0(\mathrm{p}) = S_1(\mathrm{p}) \wedge S_0(\$) = S_1(\$) \Rightarrow P(S_0) = P(S_1)$

Similarly, Q only depends on $S_0(\text{result})$ and $S_0(\$)$. By this, the remaining goal can be proved as follows:

$$
\begin{array}{l}
\quad \nu(\mathrm{y}) \wedge P(S[\text{this} := S(\mathrm{y}), \mathrm{p} := \epsilon(S, \mathrm{e})]) \wedge \mathit{nsem}(N, S, \mathrm{x{=}y.T{:}m(e);}, S'') \\
\Rightarrow \text{ [Definition of } \mathit{nsem} \text{ (disjunct for ``x=y.T:m(e);''); cf. paragraph ``SOS in HOL'']} \\
\quad \nu(\mathrm{y}) \wedge P(S[\text{this} := S(\mathrm{y}), \mathrm{p} := \epsilon(S, \mathrm{e})]) \wedge \ N > 0 \ \wedge \ \exists S' : \nu(\mathrm{y}) \wedge \tau(S(\mathrm{y})) \preceq \mathrm{T} \\
\quad \wedge \ \mathit{nsem}(N-1, \sigma, \mathit{body}(\mathit{impl}(\tau(S(\mathrm{y})), \mathrm{m})), S') \ \wedge \ S'' = S[\mathrm{x} := S'(\text{result}), \$:= S'(\$)] \\
\Rightarrow \text{ [case assumption } N > 0\text{; general logic]} \\
\quad \exists S' : S'' = S[\mathrm{x} := S'(\text{result}), \$:= S'(\$)] \\
\quad \wedge \nu(\mathrm{y}) \wedge P(S[\text{this} := S(\mathrm{y}), \mathrm{p} := \epsilon(S, \mathrm{e})]) \wedge \tau(S(\mathrm{y})) \preceq \mathrm{T} \\
\quad \wedge \ \mathit{nsem}(N-1, \sigma, \mathit{body}(\mathit{impl}(\tau(S(\mathrm{y})), \mathrm{m})), S') \\
\Rightarrow \text{ [} P(S[\text{this} := S(\mathrm{y}), \mathrm{p} := \epsilon(S, \mathrm{e})]) = P(\sigma)\text{; lemma (+)]} \\
\quad \exists S' : S'' = S[\mathrm{x} := S'(\text{result}), \$:= S'(\$)] \ \wedge \ Q(S') \\
\Rightarrow \text{ [} S'(\text{result}) = S''(\mathrm{x}) = S''[\text{result} := S''(\mathrm{x})](\text{result}),\ S'(\$) = S''[\text{result} := S''(\mathrm{x})](\$)] \\
\quad \exists S' : Q(S''[\text{result} := S''(\mathrm{x})]) \\
\Rightarrow \\
\quad Q(S''[\text{result} := S''(\mathrm{x})])
\end{array}
$$

5 Conclusions

We introduced the sequential Java subset Java-K, which provides the typical OO-language features such as classes and interfaces, subtyping, inheritance, dynamic dispatch, and encapsulation. Based on a formalization of object stores as first-order values, we presented a Hoare-style programming logic for Java-K. A central concept of this logic is the notion of virtual methods to handle overriding and dynamic dispatch. Virtual methods represent the common properties of all corresponding subtype methods. We showed how virtual methods and method implementations can be used to cover statically and dynamically bound method invocations, subtyping, and inheritance in programming logics.

The logic has been proved sound w.r.t. an SOS semantics of Java-K. Following the ideas of [Gor89], we embedded both semantics into a higher-order logic and derived the axioms and rules of the programming logic from those of the operational semantics. We presented the proofs for two typical rules. This technique for soundness proofs provides a good basis for applying proof checkers. Mechanical checking of the soundness proof is considered further work.

References

AL97. M. Abadi and K. R. M. Leino. A logic of object-oriented programs. In M. Bidoit and M. Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE, Lille, France*, volume 1214 of *Lecture Notes in Computer Science*, pages 682–696. Springer-Verlag, 1997.

BCD+89. P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: The system. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SIGPLAN Notices 24(2), 1989.

Gor89. M. J. C. Gordon. Mechanizing programming logics in higher order logic. In G. Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*. Springer-Verlag, 1989.

JvdBH+98. B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about Java classes. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 1998. Also available as TR CSI-R9812, University of Nijmegen.

Lea96. G. T. Leavens. An overview of Larch/C++: Behavioral specifications for C++ modules. In H. Kilov and W. Harvey, editors, *Specification of Behavioral Semantics in Object-Oriented Information Modeling*, chapter 8, pages 121–142. Kluwer Academic Publishers, Boston, 1996.

Lei97. K. R. M. Leino. Ecstatic: An object-oriented programming language with an axiomatic semantics. In B. Pierce, editor, *Proceedings of the Fourth International Workshop on Foundations of Object-Oriented Languages*, 1997. Available from: `www.cs.indiana.edu/hyplan/pierce/fool/`.

MPH97. P. Müller and A. Poetzsch-Heffter. Formal specification techniques for object-oriented programs. In M. Jarke, K. Pasedach, and K. Pohl, editors, *Informatik 97: Informatik als Innovationsmotor*, Informatik Aktuell. Springer-Verlag, 1997.

PH97. A. Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, Jan. 1997. `www.informatik.fernuni-hagen.de/pi5/publications.html`.

PHM98. A. Poetzsch-Heffter and P. Müller. Logical foundations for typed object-oriented languages. In D. Gries and W. De Roever, editors, *Programming Concepts and Methods (PROCOMET)*, 1998.

vON98. D. von Oheimb and T. Nipkow. Machine-checking the Java specification: Proving type-safety. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*. Springer, 1998. To appear.

Appendix

These are the SOS rules for static method invocations, constructor calls, sequential statement composition, cast, while, and if statements:

$$\frac{\mathit{initS}[\mathrm{p} := \epsilon(S,\mathrm{e}), \$:= S(\$)] : \mathit{body}(\mathrm{T@m}) \ \rightarrow S'}{S : \ \mathrm{x{=}T.m(e);} \ \rightarrow S[\mathrm{x} := S'(\mathrm{result}), \$:= S'(\$)]}$$

$$\frac{\mathit{true}}{S : \ \mathrm{x{=}new\ T();} \ \rightarrow S[\mathrm{x} := \mathrm{new}(S(\$),\mathrm{T}), \$:= S(\$)\langle\mathrm{T}\rangle]}$$

$$\frac{S : \ \mathrm{stm1} \ \rightarrow S', \ S' : \ \mathrm{stm2} \ \rightarrow S''}{S : \ \mathrm{stm1\ stm2} \ \rightarrow S''} \qquad \frac{\tau(\epsilon(S,\mathrm{e})) \preceq \mathrm{T}}{S : \ \mathrm{x{=}(T)e;} \ \rightarrow S[\mathrm{x} := \epsilon(S,\mathrm{e})]}$$

$$\frac{\epsilon(S,\mathrm{e}) = b(\mathit{true}), \ S : \ \mathrm{stm} \ \rightarrow S', \ S' : \ \mathrm{while(e)\{stm\}} \ \rightarrow S''}{S : \ \mathrm{while(e)\{stm\}} \ \rightarrow S''} \qquad \frac{\epsilon(S,\mathrm{e}) = b(\mathit{false})}{S : \ \mathrm{while(e)\{stm\}} \ \rightarrow S}$$

$$\frac{\epsilon(S,\mathrm{e}) = b(\mathit{true}), \ S : \ \mathrm{stm1} \ \rightarrow S'}{S : \ \mathrm{if(e)\{stm1\}\ else\{stm2\}} \ \rightarrow S'} \qquad \frac{\epsilon(S,\mathrm{e}) = b(\mathit{false}), \ S : \ \mathrm{stm2} \ \rightarrow S'}{S : \ \mathrm{if(e)\{stm1\}\ else\{\ stm2\ \}} \ \rightarrow S'}$$

Set-Based Failure Analysis for Logic Programs and Concurrent Constraint Programs

Andreas Podelski[1], Witold Charatonik[1⋆], and Martin Müller[2]

[1] Max-Planck-Institut für Informatik
Im Stadtwald, 66123 Saarbrücken, Germany
{podelski;witold}@mpi-sb.mpg.de
[2] Programming Systems Lab, Universität des Saarlandes
66041 Saarbrücken, Germany
mmueller@ps.uni-sb.de

Abstract. This paper presents the first approximation method of the finite-failure set of a logic program by set-based analysis. In a dual view, the method yields a type analysis for programs with ongoing behaviors (perpetual processes). Our technical contributions are (1) the semantical characterization of finite failure of logic programs over infinite trees and (2) the design and soundness proof of the first set-based analysis of logic programs with the greatest-model semantics. Finally, we exhibit the connection between finite failure and the inevitability of the 'inconsistent-store' error in fair executions of concurrent constraint programs where no process suspends forever. This indicates a potential application to error diagnosis for concurrent constraint programs
Keywords: abstract interpretation, set-based program analysis, types, logic programs, concurrent constraint programs, finite failure, fairness

1 Introduction

Set-based program analysis dates back to Reynolds [35] and Jones and Muchnick [27] and forms a well-established research topic by now (see [1,24,34] for overviews and further references). It has direct practical applications to type inference, optimization and verification of imperative, functional, logic and, as we will see in this paper, also concurrent programs.

In set-based analysis, the problem of reasoning about runtime properties of programs is transferred to the problem of solving set constraints. The design of a specific analysis involves two steps: (1) define a mapping from a class of programs P to set constraints φ_P and show the soundness of the abstraction of P by a distinguished solution of φ_P, and (2) single out a corresponding subclass of set constraints and devise an efficient algorithm for computing the distinguished solution. For instance, Heintze and Jaffar defined a set-based analysis for logic programs with the least model semantics in [22]. Their analysis is an approximation method for the *success set* of a logic program, i.e. for the set of initial queries for which a successfully terminating execution exists.

⋆ On leave from University of Wrocław, Poland. Partially supported by Polish KBN grant 8T11C02913.

S.D. Swierstra (Ed.): ESOP/ETAPS'99, LNCS 1576, pp. 177–192, 1999.
© Springer-Verlag Berlin Heidelberg 1999

In this paper, we consider the *finite failure set* of a logic program, i.e. the set of initial queries for which all fair executions terminate with failure. In order to give a *sound* prediction of finite failure ('if predicted, it will occur'), we need a characterization of finite failure in terms of program semantics. Classical results from logic programming, however, only yield the converse, i.e. a characterization of the greatest-model semantics in terms of finite failure (see Remark 1). Fortunately, for programs over the domain of infinite trees we can characterize finite failure through the greatest-model semantics (more precisely, its complement; see Theorem 1). Since the analysis we design computes an abstraction of that semantics, we obtain an approximation method for the finite-failure set of a logic program over infinite trees (see Theorem 3). More precisely, the emptiness of the computed abstract value for the predicate p indicates the finite failure of every predicate call $p(x)$. At the same time, this method can predict finite failure of a logic program over rational trees, or over finite trees (see Remarks 3 and 5).

In the least-model analysis in [22], Heintze and Jaffar use *definite* set constraints; they give a corresponding constraint solving algorithm in [21] (see [9] for further results). Our analysis uses *co-definite* set constraints, which bear their name in duality to definite set constraints due to the fact that every satisfiable constraint in this class has a greatest solution. This fact is crucial for our analysis. Algorithms for solving co-definite set constraints are given in [4,16]. In this paper, we focus on the definition of the analysis and the soundness of the abstraction, which is: the greatest solution of the co-definite set constraint φ_P inferred from the program P is a safe approximation of the greatest-model semantics for P (see Theorem 2).

In a different reading, our abstraction method is a *type analysis* of logic programs with *ongoing behavior*. Such programs are investigated under the denomination *perpetual processes* in [28]. There, the semantics of such a program P is defined by the greatest-fixpoint semantics over the domain of infinite trees. Our analysis computes the abstraction of this semantics in the form of the greatest solution of the inferred co-definite set constraint (the greatest-fixpoint semantics is equal to the greatest model of P's completion [10]). This solution assings to every program variable x a set of infinite trees that can be viewed as the *type* of x. This type describes a safe approximation (i.e. a superset) of the set of all possible runtime values for x in ongoing program executions.

Finally, we consider a potential application to *concurrent constraint* programs (see e.g. [36,37]). We carry over the approximation method of the greatest model to cc programs. This yields a type analysis for cc programs in the same sense as above. It also yields a failure analysis. In cc programs, an inconsistent constraint store (viz., failure) is considered a runtime error. (This is in contrast to logic programming where failure is part of the backtracking mechanism.) Our analysis computes an approximation of the execution states of cc programs for which failure is inevitable in fair executions unless a process (i.e. a predicate call) suspends forever (see Theorem 4). The global suspension of a process is not necessarily a programming error. That a process *must* suspend forever in order to avoid a runtime error is, however, a problem worth diagnosing and reporting.

Related Work. To our knowledge, set-based analysis for logic programming (see e.g. [5,18,13,14,22,23,30]) has previously only been designed to approximate the success set (which can be characterized by the least model semantics). Mishra's analysis [30] is often cited as the historically first one here. Heintze and Jaffar [23] have shown that Mishra's analysis is less accurate than theirs in two ways, due to the choice of the greatest solution for the class of set constraints he considers (see Remark 4) and due to the choice of the non-standard interpretation of non-empty *path-closed* sets of finite trees, respectively. Using the techniques in this paper, we are able to show that Mishra's approximation is so weak that it even approximates the greatest model. Mishra proves that '$p(x)$ will never succeed' if the set constraint ψ_P he derives is unsatisfiable. Our results yield that '$p(x)$ will finitely fail' if ψ_P is unsatisfiable over the domain of non-empty path-closed sets of *infinite* trees (see Remark 6).

Regarding the analysis of concurrent constraint programs, various techniques based on abstract interpretation have been used (see e.g. [17]) but none that is related to set-based analysis. A first formal calculus for (partial) correctness of cc programs is developed in [15]. The proof methods there are more powerful than ours but not automatic. The necessity to consider greatest-fixed point semantics for the analysis of reactive systems has been observed by other authors and in the context of different programming paradigms (see e.g. [11,19]). None of these analyses is set-based.

Finally, we want to mention that the idea to derive necessary conditions for the inevitability of a runtime error by static analysis stems from the work of Bourdoncle [3] on *abstract debugging*.

2 Logic Programs

Preliminaries. We assume a ranked alphabet Σ fixing the arity $n \geq 0$ of its function symbols $f, g, \ldots$ and constant symbols $a, b, \ldots$, and an infinite set Var of variables $x, y, z, \ldots$. We write $\bar{x}$ for finite sequences of variables, and use analogous sequence notation for other syntactic entities. We also write $f(\bar{x})$ for flat terms, where we assume implicitly that the arity of f equals the length of $\bar{x}$. A term without variables is called a *ground term*. The set of *infinite trees* over Σ is denoted by T_Σ^∞. Note that an infinite tree can have finite paths (ending with a constant symbol); a finite tree is a special case. The set of terms over Σ and Var is denoted by $T_\Sigma^\infty(\mathsf{Var})$. For an arbitrary formula Φ, we write $\exists_{-x}\Phi$ for the existential closure of Φ with respect to all variables in Φ but x. We also assume a set Pred of predicate symbols. The *Herbrand Base* $\mathcal{B}$ is the set of all ground atoms over Pred and T_Σ^∞, i.e., $\mathcal{B} = \{p(t) \mid p \in \mathsf{Pred}, t \in T_\Sigma^\infty\}$.[1]

Logic Programs. A *logic program* defines predicates through *clauses* of the form

$$p(t) \leftarrow p_1(t_1), \ldots, p_n(t_n)$$

[1] What we call Herbrand Base is sometimes called *Complete Herbrand Base* [28] in order to distinguish it from the classical notion for finite trees.

where $p(t)$ is called the *head* and $p_1(t_1), \ldots, p_n(t_n)$ is called the *body* of the clause. A clause with an empty body is called a *fact*. A complete program has the form

$$\bigwedge_{p \in \mathsf{Pred}} \bigwedge_{i=1}^{n_p} p(t_i) \leftarrow \bigwedge_{j=1}^{n_{i,p}} p_{ij}(t_{ij}) \,.$$

where i ranges over the number n_p of clauses in the definition of predicate p, and j ranges over the number $n_{i,p}$ of queries in the ith clause of predicate p. For better readability, we assume that all predicates are unary; the results can easily be extended to the case without this restriction (for example, by requiring the signature to contain at least one binary function symbol).

If we consider the logical semantics of a program of the form above, we take the *completion of* P [10], which is given by the following formula.

$$compl(P) \quad \equiv \quad \bigwedge_{p \in \mathsf{Pred}} \forall x \;\; p(x) \leftrightarrow \bigvee_{i=1}^{n_p} \exists_{-x}(\, x = t_i \wedge \bigwedge_{j=1}^{n_{i,p}} p_{ij}(t_{ij}) \,).$$

A *query* s is a conjunction $\bigwedge_k p_k(t_k)$ where the t_k are terms. We here allow infinite terms like $f(x, f(x, \ldots))$ in order to model execution states with cyclic unifiers such $y \mapsto f(x, y)$. Such terms can be finitely represented by equations, e.g. $y = f(x, y)$, or by syntact annotations as in [2].

A *ground query* is a query $\bigwedge_k p_k(t_k)$ such that all t_k are ground (i.e. without variables). We use the predicate constant *true* as the neutral element for conjunction: i.e., $s = s \wedge true$. In particular, the 'empty' query is written as *true*.

An *interpretation* ρ (sometimes called a *model*) is a subset of the Herbrand Base, $\rho \subseteq \mathcal{B}$. Interpretations are ordered by subset inclusion.

We identify an interpretation $\rho \subseteq \mathcal{B}$ with a valuation $\rho : \mathsf{Pred} \to 2^{T_\Sigma^\infty}$, i.e. a mapping of predicate symbols to sets of trees such that

$$\rho(p) = \{t \in T_\Sigma^\infty \mid p(t) \in \rho\}.$$

A *model of the program* P is a valuation $\rho : \mathsf{Pred} \to 2^{T_\Sigma^\infty}$ such that the formula $compl(P)$ is valid in the usual logical sense.

The greatest model of $compl(P)$, denoted by $gm(P)$, always exists. Using our convention of identifying the interpretation $gm(P)$ with a valuation, we use the notation $gm(P)(p)$ for the denotation of the predicate p by the greatest-model semantics, i.e.

$$gm(P)(p) = \{t \in T_\Sigma^\infty \mid p(t) \in gm(P)\}.$$

Operational Semantics. The logic program P defines a *fair transition system* $\mathcal{T}_P = \langle \mathcal{S}, \tau_P \rangle$. The fairness of the transition system is defined by the fairness of the non-deterministic *selection rule* (in the classical sense [28]: a selection rule is fair if every query atom in a state s gets selected eventually, in every execution starting in s). The non-determinism of the selection rule means that conjunction corresponds to parallel composition with the interleaving semantics; disjunction corresponds to non-deterministic choice.

The set $\mathcal{S}$ of states of the transition system $\mathcal{T}_P$ consists of all queries (including *true*) and the *failure* state *false*,

$$\mathcal{S} \quad = \quad \{\bigwedge_k p_k(t_k) \mid \forall k \;\; p_k \in \mathsf{Pred},\; t_k \in T_\Sigma^\infty(\mathsf{Var})\} \cup \{\mathit{false}\}$$

The transition relation $\tau_P \subseteq \mathcal{S} \times \mathcal{S}$ is defined according to the standard rewriting semantics under a fair selection rule. When a *selected* query atom $p(t)$ in a state $s \in \mathcal{S}$ of the form $s = s_{rest} \wedge p(t)$ unifies with the head of a clause $p(t_i) \leftarrow \bigwedge_i p_{ij}(t_{ij})$, then the state s' obtained as the instantiation of $s_{rest} \wedge \bigwedge_i p_{ij}(t_{ij})$ under the most general unifier of t and t_i is a possible successor state of s. We say that $p(t)$ is *applied* in the transition step from s to s'. When a selected query atom $p(t)$ does not unify with any of the heads of the clauses of p, then the successor state is *false*.

Similarly, P defines a fair *ground* transition system $\mathcal{T}_P^g = \langle \mathcal{S}, \tau_P^g \rangle$. We obtain the transition relation τ_P^g by modifying the one of $\mathcal{T}_P$: after every transition step of $\mathcal{T}_P^g$, all variables in the successor state are instantiated with ground terms (i.e. infinite trees). Note that ground queries are a special case of queries.

We say that a derivation *finitely fails* if if it ends in the state *false*. A query $p(x)$ is *finitely failed* (and belongs to the set FF) if every $\mathcal{T}_P$ derivation starting with query $p(x)$ finitely fails.

$$FF \quad = \quad \{p(x) \;\mid\; p(x) \text{ is finitely failed}\}$$

Similarly, a ground query $p(t)$ is called *ground finitely failed* (and belongs to the set GFF) if every T_P^g derivation starting from $p(t)$ finitely fails.

$$GFF \quad = \quad \{p(t) \;\mid\; p(t) \text{ is ground finitely failed}\}$$

We will now characterize the finite failure set of a program P over the domain T_Σ^∞ of infinite trees through the greatest model of $\mathit{compl}(P)$. Since we have not found this observation in the literature, we will give its proof, drawing from several results that are classical in the theory of logic programming.

Theorem 1 (Characterization of finite failure over infinite trees). Given a logic program P over infinite trees, the query $p(x)$ is finitely failed if and only if the value of p in the greatest model of $\mathit{compl}(P)$ over the domain T_Σ^∞ of infinite trees is the empty set; i.e.,

$$p(x) \in FF(P) \;\text{ if and only if }\; \mathit{gm}(P)(p) = \emptyset.$$

Proof. The only-if direction is a classical result (namely, the 'algebraic soundness of finite failure', see [28,25]).

For the other direction, first note that equations over infinite trees have the *saturation property*, that is, an infinite set of constraints is satisfiable if every of its finite subsets is [28,26,33].

Now assume that $p(x) \notin FF(P)$. Since (see [28,26])

$$gm(P)(p) = \{t \mid p(t) \notin GFF(P)\},$$

it is sufficient to show that there exists an infinite tree t such that $p(t) \notin GFF(P)$ (i.e., $p(t)$ is not in the ground finite failure set; note that in general, the ground finite failure of a call does not imply finite failure of some ground instance of this call.)

By assumption, there exists an execution starting in the state $p(x)$ that does not lead to the failure state. That is, there exists a transition sequence $s_0, s_1, s_2, \ldots$ starting in $s_0 = p(x)$ such that the constraint store φ_i of every state s_i is satisfiable (in the terminology of constraint logic programming [25], a state $\bigwedge_k p_k(t_k)$ is written as the pair $\langle \bigwedge_k p_k(x_k), \varphi \rangle$ where the constraint store φ is a conjunction of equations that is equivalent to $\bigwedge_k x_k = t_k$ over the domain of infinite trees). Since φ_i is stronger than φ_{i-1} for $i \geq 1$, φ_n is equivalent to $\bigwedge_{i=0}^{n} \varphi_i$.

Thus, we have a sequence of constraints $\varphi_0, \varphi_1, \varphi_2, \ldots$ such that $\bigwedge_{i=0}^{n} \varphi_i$ is satisfiable for all n. The saturation property yields that also the infinite conjunction $\bigwedge_{i \geq 0} \varphi_i$ is satisfiable. Let α be a solution of $\bigwedge_{i \geq 0} \varphi_i$. Then the transition sequence $s'_o, s'_1, s'_2, \ldots$ that we obtain by instantiating the states s_i by the valuation α is a ground transition sequence that does not lead to the fail state. Hence, if $\alpha(x) = t$, then $p(t) \notin GFF(P)$ and $GFF(P)(p)$ is nonempty. □

Remark 1. Palmgren [33] has shown that a constraint logic program over a constraint domain with the saturation property is canonical. That is, $gfp(T_P) = T_P{\downarrow}^{\omega}$ (where ${}_P{\downarrow}^{\omega} = \bigcap_{i=1}^{\omega} T_P^i(\mathcal{B})$ holds; for the definition of T_P see Section 4.) Since $gfp(T_P) = \mathcal{B} \backslash GFF(P)$ holds for canonical programs (see [25]), this is sufficient to characterize *ground* finite failure over infinite trees. Canonicity is not sufficient for finite failure of non-ground queries.

For example, consider the program $p(f(x)) \leftarrow p(x)$ over the structure of *finite* trees. This program is canonical (over finite trees). Its greatest model over finite trees assigns p the empty set (in accordance with the fact that $p(t) \in GFF(P)$ for all *finite* trees t), but $p(x)$ is not finitely failed.

Similarly, Jaffar and Stuckey [26] have shown that for programs over infinite trees, $T_P \downarrow \omega$ equals the complement of $[FF(P)]$, where $[FF(P)]$ is the set of *ground instances* of elements of $FF(P)$. This is a characterization of the denotational semantics through the operational semantics; our characterization is the converse.

Remark 2. The statement of Theorem 1 holds for constraint logic programs over every constraint system with the saturation property ('an infinite set of constraints is satisfiable if every of its finite subsets is').

Remark 3. Since the structure of *rational* trees and the structure of infinite trees are elementarily equivalent [29] (in particular, the test of satisfiability of constraints is the same), we can take the operational semantics of programs over rational trees in Theorem 1 (but we must consider the logical semantics over infinite trees; note that rational tree constraints do not have the saturation property). The modified statement is:
Given a logic program P over rational *trees, the query $p(x)$ is finitely failed if and only if the value of p in the greatest model of $compl(P)$ over the domain T^{∞}_{Σ} of* infinite *trees is the empty set.*

3 Co-definite Set Constraints

Syntax. A (general) *set expression* e is built from first-order terms, union, intersection, complement, and the projection operator [21]:

$$e \quad ::= \quad x \mid f(\overline{e}) \mid e \cup e' \mid e \cap e' \mid e^c \mid f^{-1}_{(k)}(e)$$

The projection $f^{-1}_{(k)}(e)$ is only defined if k is a positive integer smaller than the arity of f. If e does not contain the complement operator, then e is called a *positive* set expression. A (general) *set constraint* is a conjunction of inclusions of the form $e \subseteq e'$.

A *definite* set constraint [21] is a conjunction of inclusions $e_l \subseteq e_r$ between positive set expressions, where the set expressions e_r on the right hand side of $\subseteq$ are furthermore restricted to contain only variables, constants and function symbols and the intersection operator (i.e., no projection or union).

Definition 1. A *co-definite* set constraint φ is a conjunction of inclusions $e_l \subseteq e_r$ between positive set expressions, where the set expressions e_l on the left-hand side of $\subseteq$ are further restricted to contain only variables, constants, unary function symbols and the union operator (that is, no projection, intersection or terms with a function symbol of arity greater than one).

$$e_l \quad ::= \quad x \mid a \mid f(e) \qquad e_r \quad ::= \quad x \mid f(\overline{e}) \mid e \cup e' \mid e \cap e' \mid f^{-1}_{(k)}(e)$$

Semantics. We interpret set constraints over $2^{T^{\infty}_{\Sigma}}$, the domain of sets of trees over the signature Σ. That is, variables denote sets of trees, and a (set) valuation is a mapping $\alpha : \mathsf{Var} \to 2^{T^{\infty}_{\Sigma}}$. Tree constructors are interpreted as functions over sets of trees: the constant a is interpreted as $\{a\}$, and the function symbol f is interpreted as the function which maps sets $S_1, \ldots, S_n$ into the set

$$\{f(t_1, \ldots, t_n) \mid t_1 \in S_1, \ldots, t_n \in S_n\}\,.$$

The application of the projection operator for a function symbol f and the k-th argument position on a set S of trees is defined by

$$f^{-1}_{(k)}(S) = \{t \mid \exists t_1, \ldots t_n : \;\; t_k = t,\, f(t_1, \ldots, t_k, \ldots, t_n) \in S\}\,.$$

The set operators union $\cup$ and intersection $\cap$, as well as inclusion $\subseteq$ are interpreted as usual. Define the union of set valuations $\bigcup_i \alpha_i$ on variables as the pointwise union on the images of all variables; i.e., $(\bigcup_i \alpha_i)(x) = \bigcup_i \alpha_i(x)$.

The following properties hold for co-definite set constraints (see also [4]). These properties are essential for our proof in the following section to work, which shows soundness of abstraction.

Proposition 1 (Properties of co-definite set constraints).

1. Solutions of co-definite set constraints are closed under arbitrary unions. That is, the valuation $\bigcup_i \alpha_i$ is a solution if the valuations α_i, $i \in I$, are.
2. If satisfiable, every co-definite set constraint φ has a greatest solution, noted $gSol(\varphi)$.
3. Every co-definite set constraint without inclusions of the form $a \subseteq x$ is satisfiable.

Proof. The first claim is proved by case-distinction over the possible set inclusions. The second is an immediate corollary from the first one. (Note that the restriction to constants and monadic function symbols on the left hand side of an inclusion is crucial here. For instance, the set constraint $f(x, y) \subseteq f(a, a) \cup f(b, b)$ does not have a greatest solution; it has two maximal but incomparable ones.) In order to verify the third claim notice that the valuation which maps every variable into the empty set is a solution of co-definite set constraints without inclusions of the form $a \subseteq e$. □

Remark 4. Mishra [30] uses a class of set constraints with a non-standard interpretation over non-empty *path-closed* sets of finite trees to approximate the success set of a logic program. (A set of trees is path-closed if it can be recognized by a deterministic top-down tree automaton [20].) Set constraints over non-empty path-closed sets also have the properties 1. and 2. above. Due to the non-standard interpretation, this holds even if n-ary constructor terms are allowed on the left side of the inclusion. For example, the constraint $f(x, y) \subseteq f(a, a) \cup f(b, b)$ has a greatest solution over path-closed sets (which assigns both variables x and y the set $\{a, b\}$).

4 Set-based Analysis

We will next describe the inference of a co-definite set constraint φ_P from a logic program P. The intuition is as follows. A clause of the form $p(t_i) \leftarrow p_j(t_{ij})$ can be written equivalently as $p(x_i) \leftarrow x_i = t_i \wedge t_{ij} = x_{ij} \wedge p(x_{ij})$. Following the abstract interpretation framework, we abstract the semantics-defining fixpoint operator T_P by replacing the constraint $x_i = t_i \wedge t_{ij} = x_{ij}$ in its definition by the co-definite set constraint $x_i \subseteq t_i \wedge \Phi(t_{ij} \subseteq x_{ij})$; the operator Φ is defined below. The fixpoint equation for the abstract operator $T_P^{\#}$ is essentially the inferred set constraint φ_P. The soundness of the abstraction follows directly. The schema of our method (whose ingredients are Propositions 1 and Lemma 1 below) is described in an abstract setting in [12].

We next introduce the operator Φ that assigns an inclusion of the form $t \subseteq x$ a co-definite set constraint. For example, $\Phi(f(x, y) \subseteq f(a, a) \cup f(b, b))$ is essentially the conjunction of $x \subseteq f^{-1}_{(1)}(f(a, a) \cup f(b, b))$ and $y \subseteq f^{-1}_{(2)}(f(a, a) \cup f(b, b))$ which is equivalent to the conjunction of $x \subseteq a \cup b$ and $y \subseteq a \cup b$.

We introduce a fresh variable z_t for each subterm t appearing in the formula and then define the constraint $\Phi(t \subseteq x)$ for a term t and a variable x by induction on the depth of t.

$$\Phi(y \subseteq x) = y \subseteq x$$
$$\Phi(t \subseteq x) = \begin{pmatrix} z_t \subseteq x \wedge z_{t_1} \subseteq f^{-1}_{(1)}(z_t) \wedge \Phi(t_1 \subseteq z_{t_1}) \\ \dots \\ \wedge\, z_{t_n} \subseteq f^{-1}_{(n)}(z_t) \wedge \Phi(t_n \subseteq z_{t_n}) \end{pmatrix} \quad \text{for } t = f(t_1, \dots, t_n)$$

Lemma 1. If a tree valuation $\alpha : \mathsf{Var} \to T^{\infty}_{\Sigma}$ satisfies the equality $x = t$, then the set valuation $\sigma_\alpha : \mathsf{Var} \to 2^{T^{\infty}_{\Sigma}}$ defined by $\sigma_\alpha(x) = \{\alpha(x)\}$ satisfies the co-definite set constraints $x \subseteq t$ and $\Phi(t \subseteq x)$. □

We define the co-definite constraint φ_P inferred from P as follows. Here, we assume that the different clauses are renamed apart (if not, we apply α-renaming to quantified variables).

$$\varphi_P \quad \equiv \quad \bigwedge_{p \in \mathsf{Pred}} \; p \subseteq \bigcup_{i}^{n_p} t_i \;\wedge\; \bigwedge_{i}^{n_p} \bigwedge_{j}^{n_{i,p}} \Phi(t_{ij} \subseteq p_{ij})$$

Both, symbols $p \in \mathsf{Pred}$ and $x \in \mathsf{Var}$ act here as second-order variables ranging over sets of trees. In the following, when we compare an interpretation ρ of a logic program with a valuation σ of a set constraint, $\rho \subseteq \sigma$ means that $\rho(p) \subseteq \sigma(p)$ for all $p \in \mathsf{Pred}$.

Theorem 2 (Soundness of Abstraction).
For a logic program P, the greatest model of P's completion is smaller than the greatest solution of φ_P, formally $gm(P) \subseteq gSol(\varphi_P)$.

Proof. We first define an abstraction $T^{\#}_P$ of the T_P operator, and we prove that $gfp(T_P) \subseteq gfp(T^{\#}_P)$, using Lemma 1. In the second part we show that $gfp(T^{\#}_P) \subseteq gSol(\varphi_P)$, using here Proposition 1.

1. $gfp(T_P) \subseteq gfp(T^{\#}_P)$. The T_P operator maps an interpretation ρ to another one $T_P(\rho)$ where, for all $p \in \mathsf{Pred}$,

$$T_P(\rho)(p) \quad = \quad \left\{ t \in T_\Sigma \;\middle|\; \begin{array}{r} \exists \alpha : \mathsf{Var} \to T_\Sigma \;\; \exists i : \;\; t = \alpha(t_i), \\ T^{\infty}_{\Sigma}, \alpha \models \textstyle\bigwedge_j t_{ij} \in \rho(p_{ij}) \end{array} \right\} .$$

As usual, we write $\mathcal{M}, \alpha \models F$ if the formula F is valid under the interpretation with the valuation α on the structure (with the domain) $\mathcal{M}$. The greatest-model semantics and the greatest-fixpoint semantics of a program P coincide; i.e., the greatest model of P's completion is the greatest fixpoint of the operator T_P, formally $gm(P) = gfp(T_P)$ (see e.g. [28]).

The $T_P^{\#}$ operator maps an interpretation ρ to the interpretation $T_P^{\#}(\rho)$ where, for all $p \in \mathsf{Pred}$,

$$T_P^{\#}(\rho)(p) \;=\; \left\{ t \in T_\Sigma \;\middle|\; \begin{array}{l} \exists \sigma : \mathsf{Var} \to 2^{T_\Sigma^\infty},\;\; \exists i :\;\; t \in \sigma(t_i), \\ \qquad 2^{T_\Sigma^\infty}, \sigma \models \bigwedge_j \Phi(t_{ij} \subseteq x_{ij}) \wedge x_{ij} \subseteq \rho(p_{ij}) \end{array} \right\}.$$

Here, we use new variables x_{ij} as placeholders for p_{ij}. The variables $x \in \mathsf{Var}$ now range over sets of trees. The formula above is a co-definite set constraint with additional constants noted $\rho(p_{ij})$. The constant $\rho(p_{ij})$ is interpreted as the set $\rho(p_{ij})$.

Let $\rho' = T_P(\rho)$ and $\rho'' = T_P^{\#}(\rho)$. Then $\rho'(p) \subseteq \rho''(p)$ holds for all $p \in \mathsf{Pred}$. This can be seen as follows. For every tree valuation α satisfying the condition in the set comprehension for ρ', the set valuation σ_α defined by $\sigma_\alpha(x) = \{\alpha(x)\}$ satisfies the condition in the set comprehension for ρ''. Clearly, $\sigma_\alpha(t_{ij}) \subseteq \rho(p_{ij})$; we replace the inclusion $t_{ij} \subseteq \rho(p_{ij})$ by the equivalent conjunction $t_{ij} = x_{ij} \wedge x_{ij} \subseteq \rho(p_{ij})$. If σ_α satisfies the equality $t_{ij} = x_{ij}$ then also $\Phi(t_{ij} \subseteq x_{ij})$ by Lemma 1.

Hence, $T_P^{\#}$ is indeed an abstraction of T_p, and, thus, $gfp(T_P) \subseteq gfp(T_P^{\#})$. This concludes the first part of the proof.

2. $gfp(T_P^{\#}) \subseteq gSol(\varphi_P)$. In order to show that $gfp(T_P^{\#}) \subseteq gSol(\varphi_P)$, we first reformulate the definition of $T_P^{\#}$ as follows.

$$T_P^{\#}(\rho)(p) \;=\; \bigcup_{\sigma:\mathsf{Var}\to 2^{T_\Sigma^\infty}} \bigcup_i \{\sigma(t_i) \mid 2^{T_\Sigma^\infty}, \sigma \models \bigwedge_j \Phi(t_{ij} \subseteq x_{ij}) \;\wedge\; x_{ij} \subseteq \rho(p_{ij})\}$$

Fix ρ and let $\rho'' = T_P^{\#}(\rho)$.

We next exploit the fact that the solutions of co-definite set constraints are closed under arbitrary unions (Proposition 1). Hence, we can replace the union of solutions in the formula above by the greatest solution. We obtain that

$$\rho''(p) \;=\; \bigcup_i \sigma_i(t_i) \quad \text{where} \quad \sigma_i \;=\; gSol(\bigwedge_j \Phi(t_{ij} \subseteq x_{ij}) \wedge x_{ij} \subseteq \rho(p_{ij})).$$

Since all program variables are renamed apart, we have $\rho''(p) = \bigcup_i \sigma(t_i)$ where

$$\sigma \quad = \quad gSol(\bigwedge_i \bigwedge_j \Phi(t_{ij} \subseteq x_{ij}) \wedge x_{ij} \subseteq \rho(p_{ij})).$$

Thus, we have $\rho''(p) = \sigma(p)$ where

$$\sigma \quad = \quad gSol(p = \bigcup_i t_i \wedge \bigwedge_i \bigwedge_j \Phi(t_{ij} \subseteq x_{ij}) \wedge x_{ij} \subseteq \rho(p_{ij})).$$

Again, since all program variables are renamed apart,

$$\rho'' \quad = \quad gSol(\bigwedge_{p \in \mathsf{Pred}} p = \bigcup_i t_i \wedge \bigwedge_i \bigwedge_j \Phi(t_{ij} \subseteq x_{ij}) \wedge x_{ij} \subseteq \rho(p_{ij})).$$

Here, we equate the interpretation $\rho'' : \mathsf{Pred} \to 2^{T_\Sigma^\infty}$ with a valuation σ interpreting a formula with predicate symbols $p \in \mathsf{Pred}$ and tree variables $x \in \mathsf{Var}$ both ranging over sets of trees, and with constants of the form $\rho(p_{ij})$ standing for the corresponding sets. We omit any further formalization of this setting.

Let ρ_0 be any fixpoint of $T_P^\#$, i.e., $T_P^\#(\rho_0) = \rho_0$. This means that ρ_0 is a solution (the greatest one, in fact) of

$$\bigwedge_{p \in \mathsf{Pred}} p = \bigcup_i t_i \wedge \bigwedge_i \bigwedge_j \Phi(t_{ij} \subseteq x_{ij}) \wedge x_{ij} \subseteq \rho_0(p_{ij}).$$

That is, ρ_0 is a solution of φ_P. Hence, ρ_0 is smaller than the greatest solution of φ_P. This is true in particular if ρ_0 is chosen as the *greatest* fixpoint of $T_P^\#$. This concludes the second part of the proof. □

Theorem 3 (Set-based failure analysis for logic programs).
The query $p(x)$ is finitely failed in every fair execution of the logic program P if the value of p in the greatest solution (over sets of infinite trees) of the co-definite set constraint φ_P derived from P is the empty set; i.e., for all predicates $p \in \mathsf{Pred}$, if $gSol(\varphi_P)(p) = \emptyset$ then $p(x) \in FF(P)$.

Proof. We combine Theorems 2 and 1. □

A more precise formulation of the statement above is: the emptiness of the computed value for an argument variable in the i-th clause of p entails the finite failure of every predicate call of p with that clause.

Remark 5. Since the domains of infinite and rational trees are equivalent wrt. to finite failure, and failure over infinite trees implies failure over finite trees, we have the following two statements.
Given a logic program P over rational *trees [over* finite *trees], the query $p(x)$ is finitely failed if the value of p in the greatest solution over sets of* infinite *trees of the co-definite set constraint φ_P derived from P is the empty set.*

Remark 6. Essentially, the set constraint derived from a logic program P in the 'least-model' analysis of Mishra [30] is of the form

$$\psi_P \quad \equiv \quad \bigwedge_{p \in \mathsf{Pred}} p = \bigcup_i^{n_p} t_i \ \wedge \ \bigwedge_i^{n_p} \bigwedge_j^{n_{i,p}} t_{ij} \subseteq p_{ij}.$$

Instead of Lemma 1, we have the obvious fact that the set valuation σ_α defined by $\sigma_\alpha(x) = \{\alpha(x)\}$ satisfies the set constraint $x = t$ (which is equivalent to $t = x$) if the tree valuation α satisfies the tree constraint $x = t$. Since we also have the existence of greatest solutions over the domain of non-empty path-closed sets of (finite or infinite) trees (see Remark 4), the proof of Theorem 2 goes through also for ψ_P instead of φ_P, and the statements in this and the next section hold in the appropriate adaptation. One can prove that $gSol(\varphi_P) \leq gSol(\psi_P)$ (see [5]), i.e. the analysis using path-closed constraints is less accurate than the one with co-definite set constraints. Solving path-closed constraints is still an open problem (both, for least and for greatest solutions).

5 Concurrent Constraint Programs

We consider *concurrent constraint (cc) programs* (see e.g. [36,37]) in a normalized form such that we can employ a Prolog-style clausal syntax. This is a notational convention which is convenient to establish a connection to logic programming.

Furthermore, we consider only the case where constraints C are term equations $t_1 = t_2$ interpreted over infinite trees, as in the cc programming language and system Oz [32,37]. Hence, we can adopt a Prolog-like syntax and assume that every procedure p is defined either by a single fact or by several *guarded clauses* of the form

$$p(x) \leftarrow x = t \parallel p_1(t_1), \ldots, p_n(t_n).$$

In such a guarded clause, we call $x = t$ the *guard* and $p_1(t_1), \ldots, p_n(t_n)$ the *body*.

The operational semantics of a cc program P is defined through a fair transition system $\mathcal{T}_P^{\mathsf{cc}}$ as $\mathcal{T}_P$ for logic programs (again with the non-deterministic fair selection rule), with one important difference: A selected query $p(t)$ can only be applied if amongst the guarded clauses of predicate p there is one, the ith one with body $x = t_i \parallel \bigwedge_j p_{ij}(t_{ij})$, say, such that $x = t$ entails $\exists_{-x}\, x = t_i$; if this is the case in a state S, then the successor state will be $S \wedge \bigwedge_j p_{ij}(t_{ij})$ under the most general unifier of t and t_i (for a more precise definition, see e.g. [36,37]). Notice that a logic program is a special case of a cc program where all guards are trivially true, e.g. $x = x$.

Failure of cc *programs.* We next apply the approximation method of the previous section to logic programs abstracting cc programs in order to predict the behavior of the latter.

Define the logic program $\tilde{P}$ *abstracting* the cc program P by replacing the guard $\parallel$ with conjunction. It is an abstraction in the following sense.

Proposition 2. If the query $p(t)$ finitely fails in the logic program $\tilde{P}$ abstracting the cc program P then failure is inevitable in fair executions of the cc program P unless a process (i.e. a predicate call) suspends forever.

Proof. Observe that every (finite or infinite) fair computation in P in which no process suspends forever induces a fair computation in $\tilde{P}$. Namely, whenever a selected query $p(t)$ is applied with a guarded clause in P it can also be applied with the associated unguarded clause in $\tilde{P}$. This proves the claim by contraposition. □

Proposition 3 (Prediction of failure behavior of cc programs). Failure is inevitable in fair executions of the cc program P unless a process suspends forever, if the value of p in the greatest model of $compl(\tilde{P})$ over the domain T_Σ^∞ of infinite trees is the empty set.

Proof. We combine Proposition 2 and Theorem 1. □

Theorem 4 (Set-based failure analysis for cc programs).
Failure is inevitable in fair executions of the cc program P unless a process suspends forever if the value of p in the greatest solution (over sets of infinite trees) of the co-definite set constraint $\varphi_{\tilde{P}}$ derived from $\tilde{P}$ is the empty set.

Proof. We combine Theorem 3 and Proposition 3. □

6 Examples

We will give some examples to illustrate how our method of approximating greatest models with co-definite set constraints tests the inevitability of certain runtime errors. Consider the following simple stream program.

$$\mathsf{stream}([X, Y|S]) \leftarrow Y = s(s(X)),\ \mathsf{computation}(X),\ \mathsf{stream}([Y|S]).$$
$$\mathsf{main}(Z) \leftarrow \mathsf{stream}([Z|T]).$$

Suppose we know that the predicate computation makes sense only for (trees representing) odd numbers, whereas no such restriction is known for main and stream. This invariant can be expressed by the following set constraint, which may have been derived from another code fragment or externally provided by a program annotation.

$$\mathsf{computation} \subseteq s(0) \cup s(s(\mathsf{computation}))\ . \tag{1}$$

Further, we can approximate the set of non-failed computations of the program with the constraint

$$\begin{aligned} &\mathsf{stream} \subseteq cons(X, cons(Y, S)) \ \wedge \\ &X \subseteq \mathsf{computation} \ \wedge \ X \subseteq s^{-1}_{(1)}(s^{-1}_{(1)}(Y)) \ \wedge \\ &Y \subseteq cons^{-1}_{(1)}(\mathsf{stream}) \ \wedge \ S \subseteq cons^{-1}_{(2)}(\mathsf{stream}) \ \wedge \\ &\mathsf{main} \subseteq cons^{-1}_{(1)}(\mathsf{stream})\ . \end{aligned} \tag{2}$$

It is not difficult to see that the greatest solution of the conjunction of (1) and (2) assigns to the variable main (as well as to X, Y, and computation) the set of odd numbers. We obtain from this fact that, for example, the query main(0) inevitably leads to a state where computation is called with a wrong argument.

We illustrate now the necessity to consider infinite trees by another example. Consider the reactive logic program P defined by

$$p(f(x)) \leftarrow p(x).$$

The execution of the query $p(x)$ does not fail, whether the program is defined over the domain of finite or infinite trees. We derive the co-definite set constraint $\varphi_P \equiv p \subseteq f(x) \wedge x \subseteq p$. When interpreted over sets of finite trees, φ_P has as greatest solution the valuation assigning the empty set to p (and x). In the infinite tree case the greatest solution assigns to p the singleton set containing the infinite tree $f(f(f(\ldots)))$. That is, an interpretation of the derived co-definite set constraint over sets of finite trees does not admit the prediction of finite failure.

7 Conclusion

We have presented a set-based analysis of logic programs with ongoing behavior (i.e. with the greatest-fixpoint semantics). We have given a characterization of finite failure of logic programs over rational or infinite trees through the greatest model over infinite trees, and we have exhibited a connection between the inevitability of 'inconsistent-store' runtime error for cc programs and finite failure for logic programs, thus indicating a potential application to error diagnosis for cc programs.

Our 'greatest-model' set-based analysis of logic programs is interesting in its own right, as a particular instance of static analysis, and also in comparison with the 'least-model' set-based analyses of classical logic programs e.g. by Mishra [30] or by Heintze and Jaffar [22].

The practicability of our approach depends on the efficiency of the constraint solving. Succeeding the technical report [8] on which this paper and [4] are based, Devienne, Talbot and Tison [16] have given a strategy for solving co-definite set constraints which may achieve an exponential speedup. The realization of this set-based analysis for the Oz system, and its extension to reactive Oz programs with non-cc features such as cells and higher-order features is part of ongoing work. We have implemented a prototype version (with an incomplete constraint solver); experiments seem to indicate its potential usefulness for finding bugs.

One question arising from this work and the work by Cousot and Cousot in [12] is whether this set-based analysis is an instance of an abstract interpretation, i.e., whether our constraint-solving process is isomorphic to the iteration of an abstraction of the T_P fixpoint operator.

References

1. A. Aiken. Set constraints: Results, applications and future directions. In *Proceedings of the Workshop on Principles and Practice of Constraint Programming*, LNCS 874, pages 326–335. Springer-Verlag, 1994.
2. H. Aït-Kaci and A. Podelski. Functions as passive constraints in LIFE. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16 (3):1–40, 1994.
3. F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *Proceedings of the SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI'93)*, pages 46–55. ACM Press, 1993.
4. W. Charatonik and A. Podelski. Co-definite set constraints. In T. Nipkow, editor, *9th International Conference on Rewriting Techniques and Applications*, pages 211–225, Springer-Verlag. 1998.
5. W. Charatonik and A. Podelski. Directional Type Inference for Logic Programs. In G. Levi, editor, *Proceedings of the International Symposium on Static Analysis*. LNCS, pages 278–294, Springer-Verlag. 1998.
6. W. Charatonik, D.McAllester, D. Niwiński, A. Podelski and I. Walukiewicz. The Horn Mu-calculus. In V. Pratt, editor, *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science*, pages 58–69, IEEE Press. 1998.
7. W. Charatonik and A. Podelski. Set-based Analysis of Reactive Infinite-state Systems. In B. Steffen, editor, *Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. LNCS 1384, pages 358–375, Springer-Verlag. 1998.

8. W. Charatonik and A. Podelski. Set constraints for greatest models. Technical Report MPI-I-97-2-004, Max-Planck-Institut für Informatik. 1997.
9. W. Charatonik and A. Podelski. Set constraints with intersection. In G. Winskel, editor, *Twelfth Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 362–372, IEEE Press. 1997.
10. K. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, 1978.
11. P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *Proc. POPL '92*, pages 83–94. ACM Press, 1992.
12. P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Record of FPCA '95 - Conference on Functional Programming and Computer Architecture*, pages 170–181, La Jolla, California, USA, 25-28 June 1995. SIGPLAN/SIGARCH/WG2.8, ACM Press, New York, USA.
13. J. P. Gallagher, D. Boulanger, and H. Saglam. Practical model-based static analysis for definite logic programs. In *Proceedings of the 1995 International Symposium on Logic Programming*, pages 351–368.
14. J. P. Gallagher and L. Lafave. Regular approximation of computation paths in logic and functional languages. In *Proceedings of the Dagstuhl Workshop on Partial Evaluation*, pages 1–16. Springer-Verlag, February 1996.
15. F. de Boer, M. Gabbrielli, E. Marchiori, and C. Palamidessi. Proving concurrent constraint programs correct. In *Proceedings of the Twentieth ACM Symposium on Principles of Programming Languages (POPL)*, pages 98–108. ACM Press, 1994.
16. P. Devienne, J.-M. Talbot, and S. Tison. Solving classes of set constraints with tree automata. In G. Smolka, editor, *Proceedings of the Third International Conference on Principles and Practive of Constraint Programming*, pages 62–76, volume 1330 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.
17. M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Compositional Analysis for Concurrent Constraint Programming. In *Proceedings of the Eight Annual IEEE Symposium on Logic in Computer Science*, pages 210–221, 1993.
18. T. Frühwirth, E. Shapiro, M. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 300–309, July 1991.
19. K. L. S. Gasser, F. Nielson, and H. R. Nielson. Systematic realisation of control flow analyses for CML. In *Proceedings of ICFP'97*, pages 38–51. ACM Press, 1997.
20. F. Gécseg and M. Steinby. *Tree Automata*. Akademiai Kiado, 1984.
21. N. Heintze and J. Jaffar. A decision procedure for a class of set constraints (extended abstract). In *Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 42–51, 1990.
22. N. Heintze and J. Jaffar. A finite presentation theorem for approximating logic programs. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 197–209, January 1990.
23. N. Heintze and J. Jaffar. Semantic types for logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 141–156. MIT Press, 1992.
24. N. Heintze and J. Jaffar. Set constraints and set-based analysis. In *Proceedings of the Workshop on Principles and Practice of Constraint Programming*, LNCS 874, pages 281–298. Springer-Verlag, 1994.
25. J. Jaffar and M. J. Maher. Constraint Logic Programming: A Survey. *The Journal of Logic Programming*, 19/20:503–582, 1994.
26. J. Jaffar and P. J. Stuckey. Semantics of infinite tree logic programming. *Theoretical Computer Science*, 46:141–158, 1986.
27. N. D. Jones and S. S. Muchnick. Flow analysis and optimization of lisp-like structures. In *Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 244–256, January 1979.

28. J. W. Lloyd. *Foundations of Logic Programming*. Symbolic Computation. Springer-Verlag, Berlin, Germany, second, extended edition, 1987.
29. M.J. Maher. Complete Axiomatizations of the Algebras of Finite, Rational and Infinite Trees. In *Proceedings of the Symposium of Logic in Computer Science*, pages 348–457, IEEE Press, 1988.
30. P. Mishra. Towards a theory of types in Prolog. In *IEEE International Symposium on Logic Programming*, pages 289–298, 1984.
31. M. Müller. *Type Analysis for a Higher-Order Concurrent Constraint Language*. PhD thesis, Universität des Saarlandes, Technische Fakultät, D-66041 Saarbrücken, Germany, 1998.
32. The Oz System. Programming Systems Lab, Universität des Saarlandes. Available through the WWW at `http://www.ps.uni-sb.de/oz`. 1997.
33. E. Palmgren. Denotational semantics of Constraint Logic Programming – a non-standard approach. In B. Mayoh, E. Tyugu, and J. Penjam, editors, *Constraint Programming*, volume 131 of *NATO ASI Series F: Computer and System Sciences*, pages 261–288. Springer-Verlag, Berlin, Germany, 1994.
34. L. Pacholski and A. Podelski. Set Constraints: A Pearl in Research on Constraints (Tutorial Abstract). In G. Smolka, editor, *Proceedings of the Third International Conference on Principles and Practive of Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.
35. J. Reynolds. Automatic Computation of Data Set Definitions. *Information Processing*, 68, 1969.
36. V. A. Saraswat, M. Rinard, and P. Panangaden. Semantic Foundations of Concurrent Constraint Programming. In 18th *Symposium on Principles of Programming Languages*, pages 333–352. ACM Press, Jan. 1991.
37. G. Smolka. The Oz Programming Model. In *Volume 1000 of Lecture Notes in Computer Science*. Springer-Verlag, 1995.

An Idealized MetaML: Simpler, and More Expressive*

Eugenio Moggi[1], Walid Taha[2], Zine El-Abidine Benaissa[2], and Tim Sheard[2]

[1] DISI, Univ di Genova
Genova, Italy
moggi@disi.unige.it
[2] Oregon Graduate Institute
Portland, OR, USA
{walidt,benaissa,sheard}@cse.ogi.edu

Abstract. MetaML is a multi-stage functional programming language featuring three constructs that can be viewed as statically-typed refinements of the back-quote, comma, and eval of Scheme. Thus it provides special support for writing code generators and serves as a semantically-sound basis for systems involving multiple interdependent computational stages. In previous work, we reported on an implementation of MetaML, and on a reduction semantics and type-system for MetaML. In this paper, we present An Idealized MetaML (AIM) that is the result of our study of a categorical model for MetaML. An important outstanding problem is finding a type system that provides the user with a means for manipulating both open and closed *code*. This problem has eluded efforts by us and other researchers for over three years. AIM solves the issue by providing two type constructors, one *classifies* closed code and the other open code, and exploiting the way they *interact*. We point out that AIM can be verbose, and outline a possible remedy relating to the strictness of the closed code type.

1 Introduction

"If thought corrupts language, language can also corrupt thought"[1]. Staging computation into multiple steps is a well-known optimization technique used in many important algorithms, such as high-level program generation, compiled program execution, and partial evaluation. Yet few typed programming languages allow us to express staging in a natural and concise manner. MetaML was designed to fill this gap. Intuitively, MetaML has a special type for code that combines some features of both *open code*, that is, code that can contain free variables, and

* The research reported in this paper was supported by the USAF Air Materiel Command, contract #F19628-96-C-0161, NSF Grant IRI-9625462, DoD contract "Domain Specific Languages as a Carrier for Formal Methods", MURST progetto cofinanziato "Tecniche formali per la specifica, l'analisi, la verifica, la sintesi e la trasformazione di sistemi software", ESPRIT WG APPSEM.

[1] George Orwell, *Politics and the English Language*, 1946.

S.D. Swierstra (Ed.): ESOP/ETAPS'99, LNCS 1576, pp. 193–207, 1999.
© Springer-Verlag Berlin Heidelberg 1999

closed code, that is, code that contains no free variables. In a statically typed setting, open code and closed code have different properties, which we explain in the following section.

Open and Closed Code Typed languages for manipulating code fragments either have a type constructor for open code [9,6,3,11], or a type constructor for closed code [4,13]. Languages with open code types are useful in the study of partial evaluation. Typically, they provide constructs for building and combining code fragments with free variables, but do not allow the execution of such fragments. Being able to construct open fragments enables the user to force computations "under a lambda". Executing code fragments in such languages is hard because code can contain "not-yet-bound identifiers". In contrast, languages with closed code types are useful in the study of run-time (machine) code generation. Typically, they provide constructs for building and executing code fragments, but do not allow computations "under a lambda".

The importance of having both a way to construct and combine open code *and* to execute closed code within the same language can be intuitively explained in the context of Scheme. Efficient implementations of Domain-Specific or "little" languages can be developed as follows: First, build a translator from the source language to Scheme, then use eval to execute the generated Scheme code. Because such a translator will be defined by induction over the structure of the source term, it will need to return open terms when building the inside of a λ-abstraction (or any such binding construct), which can (and will often) contain free variables. For many languages, such an implementation would be almost as simple as an interpreter for the source language (especially if back-quote and comma are used), but would have almost no interpretative overhead.

MetaML MetaML [11,10] provides three constructs for manipulating open code and executing it: Brackets $\langle \_ \rangle$, Escape ~_ and Run run _. An expression $\langle e \rangle$ defers the computation of e; ~e splices the deferred expression obtained by evaluating e into the body of a surrounding Bracketed expression; and run e evaluates e to obtain a deferred expression, and then evaluates it. Note that ~e is only legal within lexically enclosing Brackets. Finally, Brackets in types such as `<int>` are read *"Code of int"*. To illustrate, consider the following interactive session:

```
-| val rec exp = fn n => fn x =>
      if n=0 then <1> else < ~x * ~(exp (n-1) x) >;
val exp = fn  : int -> <int> -> <int>

-| val exponent = fn n =>
      <fn a => ~(exp n <a>)>;
val exponent = fn  : int  -> <int -> int>

-| val cube = exponent 3;
val cube = <fn a => a * (a * (a * 1))> : <int -> int>
```

```
-| val program = <~cube 2>
val program = <(fn a => a * (a * (a * 1))) 2> : <int>

-| run program;
val it = 8 : int
```

Given an integer exponent `n` and a code fragment representing a base `x`, the function `exp` returns a code fragment representing a power. The function `exponent` is similar, but takes only an integer and returns a code fragment representing a function that takes a base and returns the power. The code fragment `cube` is the specialization of `exponent` to the power `3`. Next, we construct the code fragment `program` which is an application of the code of `cube` to the base `2`. Finally, the last declaration executes this code fragment.

Problem Unfortunately, the last declaration is not typable with the basic type system of MetaML [10]. The essence of the problem seems to be that MetaML has only one type constructor for code. Intuitively, to determine which code fragments can be executed safely, the MetaML type system must keep track of variables free in a code fragment. But there is no way for the type system to know that `program` is closed from its type, hence, a conservative approximation is made, and the term is rejected by the type system.

Contribution and Organization of this Paper In previous work [11], we reported on the implementation and applications of MetaML, and later [10] studied a reduction semantics and a type system for MetaML. However, there were still a number of drawbacks:

1. As discussed above, there is a typing problem with executing a separately-declared code fragment. While this problem is addressed in the implementation using a special typing rule for top-level declarations [12], this solution is *ad hoc*.
2. Only a call-by-value semantics could be defined for MetaML, because substitution was a partial function, only defined when variables are substituted with values.
3. The type judgment is needlessly complicated by the use of two indices. Moreover, the type system has been criticized for not being based on a standard logical system [13].

This paper describes the type system and operational semantics of An Idealized MetaML (AIM), whose design is *inspired* by a categorical model for MetaML [1]. AIM is strictly more expressive than any known typed multi-level language, and features:

1. An open code type $\langle t \rangle$, which corresponds to $\bigcirc t$ of $\lambda^{\bigcirc}$ [3] and $\langle t \rangle$ of MetaML;
2. A closed code type $[t]$, which corresponds to $\Box t$ of $\lambda^{\Box}$ [4];
3. Cross-stage persistence of MetaML;
4. A Run-With construct, generalizing Run of MetaML.

In a capsule, the model-theoretic approach has guided the design of AIM in two important ways: First, to achieve a *semantically sound* integration of Davies and Pfenning's $\lambda^{\Box}$ [4] and Davies' $\lambda^{\bigcirc}$ [3], we must use two separate type constructs, and not one, as was the case with MetaML. Second, we identified a *canonical isomorphism* between the (effect-free interpretation of the) two types $[t]$ and $[\langle t \rangle]$. This isomorphism formalized the interaction between open and closed code types, and lead us to both a generalization of Run, and to identifying a new and important (effectful) combinator that we have called $\mathsf{compile}\colon [\langle t \rangle] \to [t]$. In addition, the model-theoretic approach has suggested a number of simplifications over MetaML [10], which overcome the problems mentioned above:

1. The type system uses only one level annotation, like the $\lambda^{\bigcirc}$type system [3];
2. The level Promotion and level Demotion lemmas (cf. [10]), and the Substitution lemma, are proven in full generality and not just for the cases restricted to values. This development is crucial for a call-by-name semantics. Such a semantics seems to play an important role in the formal theory of Normalization by Evaluation and Type Directed Partial Evaluation [2];
3. The big-step semantics is defined in the style in which $\lambda^{\bigcirc}$was defined [3], and does not make explicit use of a stateful renaming function;
4. Terms have no explicit level annotations.

Furthermore, it is straightforward to extend AIM with new base types and constants, therefore it provides a general setting for investigating *staging combinators*.

In the rest of the paper, we present the type system and establish several of its syntactic properties. We give a big-step semantics of AIM, including a call-by-name variant, and prove type-safety. We present embeddings of $\lambda^{\bigcirc}$, MetaML and $\lambda^{\Box}$ into AIM. Finally, we discuss related works.

2 AIM: An Idealized MetaML

The definition of AIM's types $t \in T$ and terms $e \in E$ is parameterized with respect to a signature consisting of a set of **base types** b and **constants** c:

$$
\begin{aligned}
t \in T ::= {} & b \mid t_1 \to t_2 \mid \langle t \rangle \mid [t] \\
e \in E ::= {} & c \mid x \mid e_1\ e_2 \mid \lambda x.e \mid \langle e \rangle \mid \tilde{}e \mid \mathsf{run}\ e\ \mathsf{with}\ \{x_i = e_i | i \in m\} \mid \\
& \mathsf{box}\ e\ \mathsf{with}\ \{x_i = e_i | i \in m\} \mid \mathsf{unbox}\ e
\end{aligned}
$$

where m is a natural number, and is identified with the set of its predecessors.

The first four constructs are the standard ones in a call-by-value λ-calculus with constants. Bracket and Escape are the same as in MetaML [11,10]. Run-With

$$\Gamma \vdash c : t_c^n \qquad \Gamma \vdash x : t^n \text{ if } \Gamma\ x = t^m \text{ and } m \leq n \qquad \frac{\Gamma, x : t_1^n \vdash e : t_2^n}{\Gamma \vdash \lambda x.e : (t_1 \to t_2)^n}$$

$$\frac{\Gamma \vdash e_1 : (t_1 \to t_2)^n \quad \Gamma \vdash e_2 : t_1^n}{\Gamma \vdash e_1\ e_2 : t_2^n} \qquad \frac{\Gamma \vdash e : t^{n+1}}{\Gamma \vdash \langle e \rangle : \langle t \rangle^n} \qquad \frac{\Gamma \vdash e : \langle t \rangle^n}{\Gamma \vdash \tilde{} e : t^{n+1}}$$

$$\frac{\Gamma \vdash e_i : [t_i]^n \quad \Gamma^{+1}, \{x_i : [t_i]^n | i \in m\} \vdash e : \langle t \rangle^n}{\Gamma \vdash \mathsf{run}\ e\ \mathsf{with}\ x_i = e_i : t^n}$$

$$\frac{\Gamma \vdash e_i : [t_i]^n \quad \{x_i : [t_i]^0 | i \in m\} \vdash e : t^0}{\Gamma \vdash \mathsf{box}\ e\ \mathsf{with}\ x_i = e_i : [t]^n} \qquad \frac{\Gamma \vdash e : [t]^n}{\Gamma \vdash \mathsf{unbox}\ e : t^n}$$

Fig. 1. Typing Rules

generalizes Run of MetaML, in that it allows the use of additional variables x_i in the body of e if they satisfy certain typing requirements that are made explicit in the next section. Box-With and Unbox are not in MetaML, but are motivated by $\lambda^{\Box}$of Davies and Pfenning [4]. We use some abbreviated forms:

$$\begin{aligned} \mathsf{run}\ e &\ \textit{for}\ \mathsf{run}\ e\ \mathsf{with}\ \emptyset \\ \mathsf{box}\ e &\ \textit{for}\ \mathsf{box}\ e\ \mathsf{with}\ \emptyset \\ \mathsf{run}\ e\ \mathsf{with}\ x_i = e_i &\ \textit{for}\ \mathsf{run}\ e\ \mathsf{with}\ \{x_i = e_i | i \in m\} \\ \mathsf{box}\ e\ \mathsf{with}\ x_i = e_i &\ \textit{for}\ \mathsf{box}\ e\ \mathsf{with}\ \{x_i = e_i | i \in m\} \end{aligned}$$

2.1 Type System

An AIM typing judgment has the form $\Gamma \vdash e : t^n$, where $t \in T$, $n \in N$ and Γ is a type assignment, that is, a finite set $\{x_i : t_i^{n_i} | i \in m\}$ with the x_i distinct. The reading of $\Gamma \vdash e : t^n$ is *"term e has type t at level n in the type assignment Γ"*. The *level* of a subterm is the number of surrounding Brackets, minus the number of surrounding Escapes. If not otherwise indicated, the level of a term is zero. We say that $\Gamma\ x = t^n$ if $x : t^n$ is in Γ. Furthermore, we write Γ^{+r} for the type assignment obtained by incrementing the level annotations in Γ by r, that is, $\Gamma^{+r}\ x = t^{n+r}$ if and only if $\Gamma\ x = t^n$. Figure 1 gives the typing rules for AIM. The Constant rule says that a constant c of type t_c, which has to be given in the signature, can be used at any level n. The Variable rule incorporates cross-stage persistence, therefore if x is introduced at level m it can be used later, that is, at level $n \geq m$, but not before. The Abstraction and Application rules are standard. The Bracket and Escape rules establish an *isomorphism* between t^{n+1} and $\langle t \rangle^n$. Typing Run in MetaML [10] introduces an extra index-annotation on types for counting the number of Runs surrounding an expression (see Figure 3). We avoid this extra annotation by incrementing the level of all variables in Γ. In particular, the Run rule of MetaML becomes

$$\frac{\Gamma^{+1} \vdash e : \langle t \rangle^n}{\Gamma \vdash \mathsf{run}\ e : t^n}$$

The Box rule ensures that there are no "late" free variables in the term being Boxed. This ensures that when a Boxed term is evaluated, the resulting value is a closed term. The Box rule ensures that only With-bound variables can occur free in the term e. At the same time, it ensures that no "late" free variable can infiltrate the body of a Box through a With-bound variable. This is accomplished by forcing the With-bound variables themselves to have a Boxed type. Note that in $\mathsf{run}\ e\ \mathsf{with}\ x_i = e_i$ the term e may contain other free variables besides the x_i.

2.2 Properties of the Type System

The following level Promotion, level Demotion and Substitution lemmas are needed for proving Type Preservation.

Lemma 1 (Promotion). *If* $\Gamma_1, \Gamma_2 \vdash e : t^n$ *then* $\Gamma_1, \Gamma_2^{+1} \vdash e : t^{n+1}$.

Meaning that if we increment the level of a well-formed term e it remains well-formed. Furthermore, we can simultaneously increment the level of an arbitrary subset of the variables in the environment. In this paper, proofs are omitted for brevity (Please see technical report for proof details [8]).

Demotion on e at n, written $e{\downarrow_n}$, lowers the level of e from level $n+1$ down to level n, and is well-defined on all terms, unlike demotion for MetaML [10].

Definition 1 (Demotion). $e{\downarrow_n}$ *is defined by induction on* e*:*

$$
\begin{aligned}
c{\downarrow_n} &= c \\
x{\downarrow_n} &= x \\
(e_1\ e_2){\downarrow_n} &= e_1{\downarrow_n}\ \ e_2{\downarrow_n} \\
(\lambda x.e){\downarrow_n} &= \lambda x.e{\downarrow_n} \\
\langle e \rangle{\downarrow_n} &= \langle e{\downarrow_{n+1}} \rangle \\
\tilde{}e{\downarrow_0} &= \mathsf{run}\ e \\
(\tilde{}e){\downarrow_{n+1}} &= \tilde{}(e{\downarrow_n}) \\
(\mathsf{run}\ e\ \mathsf{with}\ x_i = e_i){\downarrow_n} &= \mathsf{run}\ e{\downarrow_n}\ \ \mathsf{with}\ x_i = e_i{\downarrow_n} \\
(\mathsf{box}\ e\ \mathsf{with}\ x_i = e_i){\downarrow_n} &= \mathsf{box}\ e\ \mathsf{with}\ x_i = e_i{\downarrow_n} \\
(\mathsf{unbox}\ e){\downarrow_n} &= \mathsf{unbox}\ e{\downarrow_n}
\end{aligned}
$$

The key for making demotion total on all terms is handling the case for Escape $\tilde{}e{\downarrow_0}$: Escape is simply replaced by Run. It should also be noted that demotion does not go into the body of Box.

Lemma 2 (Demotion). *If* $\Gamma^{+1} \vdash e : t^{n+1}$ *then* $\Gamma \vdash e{\downarrow_n} : t^n$.

Meaning that if we demote a well-formed term e it remains well-formed, provided the level of all free variables is decremented.

Lemma 3 (Weakening). *If* $\Gamma_1, \Gamma_2 \vdash e_2 : t_2^n$ *and* x *is* fresh, *then* $\Gamma_1, x : t_1^{n'}, \Gamma_2 \vdash e_2 : t_2^n$.

Lemma 4 (Substitution). *If* $\Gamma_1 \vdash e_1 : t_1^{n'}$ *and* $\Gamma_1, x : t_1^{n'}, \Gamma_2 \vdash e_2 : t_2^n$ *then* $\Gamma_1, \Gamma_2 \vdash e_2[x := e_1] : t_2^n$.

This is the expected substitution property, that is, a variable x can be replaced by a term e_1, provided e_1 meets the type requirements on x.

3 Big-Step Semantics

The big-step semantics for MetaML [11] reflects the existing implementation: it is complex, and hence not very suitable for formal reasoning. Figure 2 presents a concise big-step semantics for AIM, which is presented at the same level of abstraction as that for $\lambda^{\bigcirc}$ [3]. We avoid the explicit use of a *gensym* or *newname* for renaming bound variables, which here is implicitly done by substitution.

Definition 2 (Values).

$$
\begin{array}{rcll}
v^0 & \in & V^0 & ::= c \mid \lambda x.e \mid \langle v^1 \rangle \mid \mathsf{box}\ e \\
v^1 & \in & V^1 & ::= c \mid x \mid v^1\ v^1 \mid \lambda x.v^1 \mid \langle v^2 \rangle \mid \mathsf{run}\ v^1\ \mathsf{with}\ x_i = v_i^1 \mid \\
 & & & \quad \mathsf{box}\ e\ \mathsf{with}\ x_i = v_i^1 \mid \mathsf{unbox}\ v^1 \\
v^{n+2} & \in & V^{n+2} & ::= c \mid x \mid v^{n+2}\ v^{n+2} \mid \lambda x.v^{n+2} \mid \langle v^{n+3} \rangle \mid \tilde{}v^{n+1} \mid \\
 & & & \quad \mathsf{run}\ v^{n+2}\ \mathsf{with}\ x_i = v_i^{n+2} \mid \mathsf{box}\ e\ \mathsf{with}\ x_i = v_i^{n+2} \mid \mathsf{unbox}\ v^{n+2}
\end{array}
$$

Values have three important properties: First, a value at level 0 can be a Bracketed or a Boxed expression, reflecting the fact that terms representing open and closed code are both considered acceptable results from a computation. Second, values at level $n+1$ can contain Applications such as $\langle (\lambda y.y)\ (\lambda x.x) \rangle$, reflecting the fact that computations at these levels can be deferred. Finally, there are no level 1 Escapes in values, reflecting the fact that having such an Escape in a term would mean that evaluating the term has not yet been completed. This is true, for example, in terms like $\langle \tilde{}(f\ x) \rangle$.

Lemma 5 (Orthogonality). *If* $v \in V^0$ *and* $\Gamma \vdash v : [t]^0$ *then* $\emptyset \vdash v : [t]^0$.

Theorem 1 (Type Preservation). *If* $\Gamma^{+1} \vdash e : t^n$ *and* $e \overset{n}{\hookrightarrow} v$ *then* $v \in V^n$ *and* $\Gamma^{+1} \vdash v : t^n$.

Note that in AIM (unlike ordinary programming languages) we cannot restrict the evaluation rules to closed terms, because at levels above 0 evaluation is *symbolic* and can go inside the body of binding constructs. On the other hand, evaluation of a variable at level 0 is an error! The above theorem strikes the right balance, namely it allows open terms provided their free variables are at level above 0 (this is reflected by the use of Γ^{+1} in the typing judgment).

Having no level 1 Escapes ensures that demotion is the identity on V^{n+1} as shown in the following lemma. Thus, we don't need to perform demotion in the evaluation rule for Run when evaluating a well-formed term.

Evaluation.

$$\frac{e_1 \overset{0}{\hookrightarrow} \lambda x.e \quad e_2 \overset{0}{\hookrightarrow} v_1 \quad e[x:=v_1] \overset{0}{\hookrightarrow} v_2}{e_1\ e_2 \overset{0}{\hookrightarrow} v_2} \qquad \lambda x.e \overset{0}{\hookrightarrow} \lambda x.e$$

$$\frac{e_i \overset{0}{\hookrightarrow} v_i \quad e[x_i:=v_i] \overset{0}{\hookrightarrow} \langle v' \rangle \quad v' \downarrow_0 \overset{0}{\hookrightarrow} v}{\mathsf{run}\ e\ \mathsf{with}\ x_i = e_i \overset{0}{\hookrightarrow} v} \qquad \frac{e \overset{0}{\hookrightarrow} \langle v \rangle}{\tilde{}e \overset{1}{\hookrightarrow} v}$$

$$\frac{e_i \overset{0}{\hookrightarrow} v_i}{\mathsf{box}\ e\ \mathsf{with}\ x_i = e_i \overset{0}{\hookrightarrow} \mathsf{box}\ e[x_i:=v_i]} \qquad \frac{e \overset{0}{\hookrightarrow} \mathsf{box}\ e' \quad e' \overset{0}{\hookrightarrow} v}{\mathsf{unbox}\ e \overset{0}{\hookrightarrow} v}$$

Building.

$$\frac{e \overset{n+1}{\hookrightarrow} v}{\mathsf{unbox}\ e \overset{n+1}{\hookrightarrow} \mathsf{unbox}\ v} \qquad \frac{e \overset{n+1}{\hookrightarrow} v}{\lambda x.e \overset{n+1}{\hookrightarrow} \lambda x.v} \qquad x \overset{n+1}{\hookrightarrow} x$$

$$\frac{e \overset{n+1}{\hookrightarrow} v \qquad e_i \overset{n+1}{\hookrightarrow} v_i}{\mathsf{run}\ e\ \mathsf{with}\ x_i = e_i \overset{n+1}{\hookrightarrow} \mathsf{run}\ v\ \mathsf{with}\ x_i = v_i} \qquad \frac{e \overset{n+1}{\hookrightarrow} v}{\tilde{}e \overset{n+2}{\hookrightarrow} \tilde{}v} \qquad \frac{e \overset{n+1}{\hookrightarrow} v}{\langle e \rangle \overset{n}{\hookrightarrow} \langle v \rangle}$$

$$\frac{e_i \overset{n+1}{\hookrightarrow} v_i}{\mathsf{box}\ e\ \mathsf{with}\ x_i = e_i \overset{n+1}{\hookrightarrow} \mathsf{box}\ e\ \mathsf{with}\ x_i = v_i} \qquad \frac{e_1 \overset{n+1}{\hookrightarrow} v_1 \quad e_2 \overset{n+1}{\hookrightarrow} v_2}{e_1\ e_2 \overset{n+1}{\hookrightarrow} v_1\ v_2} \qquad c \overset{n+1}{\hookrightarrow} c$$

Stuck.

$$\frac{e \overset{0}{\hookrightarrow} v \not\equiv \mathsf{box}\ e'}{\mathsf{unbox}\ e \overset{0}{\hookrightarrow} err} \qquad \frac{e_1 \overset{0}{\hookrightarrow} v \not\equiv \lambda x.e}{e_1 e_2 \overset{0}{\hookrightarrow} err} \qquad x \overset{0}{\hookrightarrow} err$$

$$\frac{e_i \overset{0}{\hookrightarrow} v_i \quad e[x_i:=v_i] \overset{0}{\hookrightarrow} v \not\equiv \langle e' \rangle}{\mathsf{run}\ e\ \mathsf{with}\ x_i = e_i \overset{0}{\hookrightarrow} err} \qquad \frac{e \overset{0}{\hookrightarrow} v \not\equiv \langle e' \rangle}{\tilde{}e \overset{1}{\hookrightarrow} err} \qquad \tilde{}e \overset{0}{\hookrightarrow} err$$

Fig. 2. Big-Step Semantics

Lemma 6 (Value Demotion). *If $v \in V^{n+1}$ then $v \downarrow_n \equiv v$.*

A good property for multi-level languages is the existence of a bijection between programs $\emptyset \vdash e : t^0$ and program representations $\emptyset \vdash \langle v \rangle : \langle t \rangle^0$. This property holds for AIM. In fact it is a consequence of the following result:

Proposition 1 (Reflection). *If $\Gamma \vdash e : t^n$, then $\Gamma^{+1} \vdash e : t^{n+1}$ and $e \in V^{n+1}$. Conversely, if $v \in V^{n+1}$ and $\Gamma^{+1} \vdash v : t^{n+1}$, then $\Gamma \vdash v : t^n$.*

3.1 Call-by-Name

The difference between the call-by-name semantics and the call-by-value semantics for AIM is only in the evaluation rule for Application at level 0. For call-by-name, this rule becomes

$$\frac{e_1 \overset{0}{\hookrightarrow} \lambda x.e \quad e[x := e_2] \overset{0}{\hookrightarrow} v}{e_1\ e_2 \overset{0}{\hookrightarrow} v}$$

The Type Preservation proof need only be changed for the Application case. This is not problematic, since the Substitution Lemma for AIM has no *value restriction*.

Theorem 2 (CBN Type Preservation). *If* $\Gamma^{+1} \vdash e : t^n$ *and* $e \overset{n}{\hookrightarrow} v$ *then* $v \in V^n$ *and* $\Gamma^{+1} \vdash v : t^n$.

3.2 Expressiveness

MetaML's type system has one Code type constructor, which tries to combine the features of the Box and Circle type constructors of Davies and Pfenning. This *combination* leads to the typing problem discussed in the introduction. In contrast, AIM's type system incorporates both Box and Circle type constructors, thereby providing *correct semantics* for the following natural and desirable functions:

1. $\mathsf{unbox} : [t] \to t$. This function executes closed code. AIM has no function of the opposite type $t \to [t]$, thus we avoid the "collapse" of types in the recent work of Wickline, Lee, and Pfenning [13]. Such a function does not exist in MetaML.
2. $\mathsf{up} : t \to \langle t \rangle$. This function corresponds to cross-stage persistence [11], in fact it embeds any value into an open fragment, including values of functional type. Such a function does not exist in $\lambda^{\bigcirc}$. At the same time, AIM has no function of the opposite type $\langle t \rangle \to t$, reflecting the fact that open code cannot be executed. up is expressible as $\lambda x.\langle x \rangle$.
3. $\mathsf{weaken} : [t] \to \langle t \rangle$. This is (almost) the composite of the two functions above. weaken reflects the fact that closed code can always be viewed as open code. AIM has no function of the opposite type $\langle t \rangle \to [t]$.
4. $\mathsf{compile} : [\langle t \rangle] \to [t]$. This function allows us to convert a Boxed Bracket value into a Boxed value. It can be viewed as the essence of the interaction between the Bracket and the Box type. Compile is not expressible (with the desired strictness behavior) in the language, but has the following operational semantics:

$$\frac{e \overset{0}{\hookrightarrow} \mathsf{box}\ e' \quad e' \overset{0}{\hookrightarrow} \langle v' \rangle}{\mathsf{compile}\ e \overset{0}{\hookrightarrow} \mathsf{box}\ (v' \downarrow_0)}\ .$$

Type Preservation is still valid with such an extension.

5. execute: $[\langle t \rangle] \to t$. This function executes closed code. It can be defined in terms of Run-With as $\lambda x.\mathsf{run}\ \mathsf{unbox}\ x\ \mathsf{with}\ x = x$, and also in terms of Compile as $\lambda x.\mathsf{unbox}\ (\mathsf{compile}\ x)$.

Now, the MetaML example presented in the Introduction can be expressed in AIM as follows:

```
-| val rec exp = box (fn n => fn x =>
      if n=0 then <1> else < ~x * ~((unbox exp) (n-1) x) >)
      with {exp=exp};
val exp = [fn]  : [int -> <int> -> <int>]

-| val exponent = box (fn n =>
      <fn a => ~((unbox exp) n <a>)>)
      with {exp=exp};
val exponent = [fn]  : [int  -> <int -> int>]

-| val cube = compile (box ((unbox exponent) 3)
                       with {exponent=exponent});
val cube = [fn a => a * (a * (a * 1))] : [int -> int]

-| val program = compile(box <(unbox cube) 2>
                         with {cube=cube})
val program = [(fn a => a * (a * (a * 1))) 2] : [int]

-| unbox program;
val it = 8 : int
```

In AIM, asserting that a code fragment is closed (using Box) has become part of the responsibilities of the programmer. Furthermore, Compile is needed to explicitly overcome the default lazy behavior of Box. If Compile was not used in the above examples, the (Boxed code) values returned for `cube` and `program` would contain unevaluated expressions.

Unfortunately, the syntax is verbose compared to that of MetaML. In future work, we hope to improve the syntax based on experience using AIM. In particular, we plan to investigate an eager operational semantics for Box, which should simplify the *formalization* of MetaML constructs in AIM, and perhaps make the Compile combinator unnecessary.

4 Embedding Results

This section shows that other languages for staging computations can be translated into AIM, and that the embedding *respects* the typing and evaluation. The languages we consider are $\lambda^{\bigcirc}$ [3], MetaML [10], and $\lambda^{\Box}$ [4].

4.1 Embedding of $\lambda^{\bigcirc}$

The embedding of $\lambda^{\bigcirc}$ into AIM is straightforward. In essence, $\lambda^{\bigcirc}$ corresponds to the **Open fragment** of AIM:

$$t \in T_{Open} ::= b \mid t_1 \to t_2 \mid \langle t \rangle$$
$$e \in E_{Open} ::= c \mid x \mid e_1\ e_2 \mid \lambda x.e \mid \langle e \rangle \mid \tilde{}\, e$$

The translation $(\_^{\bigcirc})$ between $\lambda^{\bigcirc}$and AIM is as follows: $(\bigcirc t)^{\bigcirc} = \langle (t^{\bigcirc}) \rangle$, $(\mathsf{next}\ e)^{\bigcirc} = \langle e^{\bigcirc} \rangle$, and $(\mathsf{prev}\ e)^{\bigcirc} = \tilde{}\, (e^{\bigcirc})$. With these identifications the typing and evaluation rules for $\lambda^{\bigcirc}$ are those of AIM restricted to the relevant fragment. The only exception is the typing rule for variables, which in $\lambda^{\bigcirc}$ is simply $\Gamma \vdash x : t^n$ if $\Gamma\ x = t^n$ (this reflects the fact that $\lambda^{\bigcirc}$ has no cross-stage persistence).

We write $\Gamma \vdash_{\bigcirc} e : t$ and $e \overset{n}{\hookrightarrow}_{\bigcirc} v$ for the typing and evaluation judgments of $\lambda^{\bigcirc}$, so that they are not confused with the corresponding judgments of AIM.

Proposition 2 (Temporal Type Embedding). *If $\Gamma \vdash_{\bigcirc} e : t^n$ is derivable in $\lambda^{\bigcirc}$, then $\Gamma^{\bigcirc} \vdash e^{\bigcirc} : (t^{\bigcirc})^n$ is derivable in AIM.*

Proposition 3 (Temporal Semantics Embedding). *If $e \overset{n}{\hookrightarrow}_{\bigcirc} v$ is derivable in $\lambda^{\bigcirc}$, then $e^{\bigcirc} \overset{n}{\hookrightarrow} v^{\bigcirc}$ is derivable in AIM.*

4.2 Embedding of MetaML

The difference between MetaML and AIM is in the type system. We show that while AIM's typing judgments are simpler, what is typable in MetaML remains typable in AIM.

$$t \in T_{MetaML} ::= b \mid t_1 \to t_2 \mid \langle t \rangle$$
$$e \in E_{MetaML} ::= c \mid x \mid e_1\ e_2 \mid \lambda x.e \mid \langle e \rangle \mid \tilde{}\, e \mid \mathsf{run}\ e$$

MetaML's typing judgment has the form $\Delta \vdash_M e : (t, r)^n$, where $t \in T$, $n, r \in N$ and Δ is a type assignment, that is, a finite set $\{x_i : (t_i, r_i)^{n_i} | i \in m\}$ with the x_i distinct. We use the subscript M to distinguish MetaML's judgments from AIM's judgments. Figure 3 recalls the type system of MetaML [10].

Definition 3 (Acceptable Judgment). *We say that a MetaML typing judgment $\{x_i : (t_i, r_i)^{n_i} | i \in m\} \vdash_M e : (t, r)^n$ is* **acceptable** *if and only if $\forall i \in m.\ r_i \leq r$.*

Remark 1. A careful analysis of MetaML's typing rules shows that typing judgments occurring in the derivation of a judgment $\emptyset \vdash_M e : (t, r)^n$ are acceptable. In fact in a MetaML typing rule the premises are acceptable whenever its conclusion is acceptable, simply because the index r never decreases when we go from the conclusion of a type rule to its premises, Thus, we never get an environment binding with an r higher than that of the judgment.

$$\Gamma \vdash_M c\colon (t_c, r)^n \qquad \Gamma \vdash_M x\colon (t, r)^n \text{ if } \Gamma\ x = (t, p)^m \text{ and } m + r \leq n + p$$

$$\frac{\Gamma, x\colon (t_1, r)^n \vdash_M e\colon (t_2, r)^n}{\Gamma \vdash_M \lambda x.e\colon (t_1 \to t_2, r)^n} \qquad \frac{\Gamma \vdash_M e_1\colon (t_1 \to t_2, r)^n \quad \Gamma \vdash_M e_2\colon (t_1, r)^n}{\Gamma \vdash_M e_1\ e_2\colon (t_2, r)^n}$$

$$\frac{\Gamma \vdash_M e\colon (t, r)^{n+1}}{\Gamma \vdash_M \langle e\rangle\colon (\langle t\rangle, r)^n} \qquad \frac{\Gamma \vdash_M e\colon (\langle t\rangle, r)^n}{\Gamma \vdash_M \tilde{}e\colon (t, r)^{n+1}} \qquad \frac{\Gamma \vdash_M e\colon (\langle t\rangle, r+1)^n}{\Gamma \vdash_M \mathsf{run}\ e\colon (t, r)^n}$$

Fig. 3. MetaML Typing rules

Proposition 4 (MetaML Type Embedding). *If* $\{x_i\colon (t_i, r_i)^{n_i} | i \in m\} \vdash_M e\colon (t, r)^n$ *is acceptable, then it is derivable in MetaML if and only if* $\{x_i\colon t_i^{n_i + r - r_i} | i \in m\} \vdash e\colon t^n$ *is derivable in AIM.*

4.3 Embedding of $\lambda^{\Box}$

Figure 4 summarizes the language $\lambda^{\Box}$ [4]. We translate $\lambda^{\Box}$ into the **Closed fragment** of AIM:

$$t \in T_{Closed} ::= b \mid t_1 \to t_2 \mid [t]$$
$$e \in E_{Closed} ::= c \mid x \mid e_1\ e_2 \mid \lambda x.e \mid \mathsf{box}\ e\ \mathsf{with}\ x_i = e_i \mid \mathsf{unbox}\ e$$

We need only consider typing judgments of the form $\{x_i\colon t_i^0 | i \in m\} \vdash e\colon t^0$ and evaluation judgments of the form $e \overset{0}{\hookrightarrow} v$. These restrictions are possible for two reasons. If the conclusion of a typing rule is of the form $\{x_i\colon t_i^0 | i \in m\} \vdash e\colon t^0$ with types and terms in the Closed fragment, then also the premises of the typing rule enjoy such properties. When e is a closed term in the Closed fragment, the only judgments $e' \overset{n}{\hookrightarrow} v'$ that can occur in the derivation of $e \overset{0}{\hookrightarrow} v$ are such that $n = 0$ and e' and v' are closed terms in the Closed fragment.

Definition 4 (Modal Type Translation). *The translation of* $\lambda^{\Box}$ *types is given by*

$$b^{\Box} = b \qquad (t_1 \to t_2)^{\Box} = t_1^{\Box} \to t_2^{\Box} \qquad (\Box t)^{\Box} = [t^{\Box}]$$

The translation of $\lambda^{\Box}$ *terms depends on a set* X *of variables, namely those declared in the modal context* Δ*.*

$$\begin{aligned}
x^{\Box X} &= \mathsf{unbox}\ x \qquad \text{if } x \in X \\
y^{\Box X} &= y \qquad \text{if } y \notin X \\
(\mathsf{box}\ e)^{\Box X} &= \mathsf{box}\ e^{\Box X}\ \mathsf{with}\ \{x = x | x \in \mathrm{FV}(e) \cap X\} \\
(\mathsf{let\ box}\ x = e_1\ \mathsf{in}\ e)^{\Box X} &= (\lambda x.e^{\Box X \cup \{x\}})\ e_1^{\Box X} \\
(\lambda y.e)^{\Box X} &= \lambda y.e^{\Box X} \quad \text{where } y \notin X \\
(e_1\ e_2)^{\Box X} &= e_1^{\Box X}\ e_2^{\Box X}
\end{aligned}$$

Syntax

$$\begin{aligned} \text{Types } t \in T_\Box ::= {} & b \mid t_1 \to t_2 \mid \Box t \\ \text{Expressions } e \in E_\Box ::= {} & x \mid \lambda x.e \mid e_1\ e_2 \mid \mathsf{box}\ e \mid \mathsf{let\ box}\ x = e_1\ \mathsf{in}\ e_2 \\ \text{Type assignments } \Gamma, \Delta ::= {} & \{x_i : t_i | i \in m\} \end{aligned}$$

Type System

$$\Delta; \Gamma \vdash_\Box x : t \text{ if } \Delta x = t \qquad \Delta; \Gamma \vdash_\Box x : t \text{ if } \Gamma x = t$$

$$\frac{\Delta; (\Gamma, x:t') \vdash_\Box e : t}{\Delta; \Gamma \vdash_\Box \lambda x.e : t' \to t} \qquad \frac{(\Delta; x:t'), \Gamma \vdash_\Box e_2 : t \quad \Delta; \Gamma \vdash_\Box e_1 : \Box t'}{\Delta; \Gamma \vdash_\Box \mathsf{let\ box}\ x = e_1\ \mathsf{in}\ e_2 : t}$$

$$\frac{\Delta; \Gamma \vdash_\Box e_1 : t' \to t \quad \Delta; \Gamma \vdash_\Box e_2 : t'}{\Delta; \Gamma \vdash_\Box e_1\, e_2 : t} \qquad \frac{\Delta; \emptyset \vdash_\Box e : t}{\Delta; \Gamma \vdash_\Box \mathsf{box}\ e : \Box t}$$

Big-Step Semantics

$$\frac{e_1 \hookrightarrow_\Box \lambda x.e \quad e_2 \hookrightarrow_\Box v' \quad e[x := v'] \hookrightarrow_\Box v}{e_1,\, e_2 \hookrightarrow_\Box v} \qquad \lambda x.e \hookrightarrow_\Box \lambda x.e$$

$$\frac{e_1 \hookrightarrow_\Box \mathsf{box}\ e \quad e_2[x := e] \hookrightarrow_\Box v}{\mathsf{let\ box}\ x = e_1\ \mathsf{in}\ e_2 \hookrightarrow_\Box v} \qquad \mathsf{box}\ e \hookrightarrow_\Box \mathsf{box}\ e$$

Fig. 4. Description of $\lambda^\Box$

Proposition 5 (Modal Type Embedding). *If $\Delta; \Gamma \vdash_\Box e : t$ is derivable in $\lambda^\Box$, then $[\Delta^\Box], \Gamma^\Box \vdash e^{\Box X} : t^\Box$ is derivable in AIM's Closed fragment, where X is the set of variables declared in Δ, $\{x_i : t_i | i \in m\}^\Box$ is $\{x_i : t_i^\Box | i \in m\}$, and $[\{x_i : t_i | i \in m\}]$ is $\{x_i : [t_i] | i \in m\}$.*

The translation of $\lambda^\Box$ into the AIM's Closed fragment does not preserve evaluation *on the nose* (that is, up to syntactic equality). Therefore, we need to consider an *administrative* reduction.

Definition 5 (Box-Reduction). *The $\to_{box}$ reduction is given by the rewrite rules*

$$\mathsf{unbox}\ (\mathsf{box}\ e) \to e$$
$$\mathsf{box}\ e' \ \mathsf{with}\ x_i = e_i, x = \mathsf{box}\ e, x_j = e_j \to \mathsf{box}\ e'[x := \mathsf{box}\ e] \ \mathsf{with}\ x_i = e_i, x_j = e_j$$

where e is a closed term of the Closed fragment.

Lemma 7 (Properties of Box-Reduction). *The $\to_{box}$ reduction on the Closed fragment satisfies the following properties:*

– *Subject Reduction, that is, $\Gamma \vdash e : t$ and $e \to_{box} e'$ imply $\Gamma \vdash e' : t$*

– *Confluence and Strong Normalization*
– *Compatibility with Evaluation on closed terms, that is,* $e_1 \overset{0}{\hookrightarrow} v_1$ *and* $e_1 \xrightarrow{*}_{box} e_2$ *imply that exists* v_2 *s.t.* $v_1 \xrightarrow{*}_{box} v_2$ *and* $e_2 \overset{0}{\hookrightarrow} v_2$.

Lemma 8 (Substitutivity). *Given a closed term* $e_0 \in E_\Box$ *the following properties hold:*

– $e^{\Box X}[y := e_0^{\Box\emptyset}] \equiv (e[y := e_0])^{\Box X}$, *provided* $y \notin X$
– $e^{\Box X \cup \{x\}}[x := \mathsf{box}\ e_0^{\Box\emptyset}] \xrightarrow{*}_{box} (e[x := e_0])^{\Box X}$

Proposition 6 (Modal Semantics Embedding). *If* $e \in E_\Box$ *is closed and* $e \hookrightarrow_\Box v$ *is derivable in* $\lambda^\Box$, *then there exists* v' *such that* $e^{\Box\emptyset} \overset{0}{\hookrightarrow} v'$ *and* $v' \xrightarrow{*}_{box} v^{\Box\emptyset}$.

5 Related Work

Multi-stage programming techniques have been studied and used in a wide variety of settings [11]. Nielson and Nielson present a seminal detailed study into a two-level functional programming language [9]. Davies and Pfenning show that a generalization of this language to a multi-level language called $\lambda^\Box$ gives rise to a type system related to a modal logic, and that this type system is equivalent to the binding-time analysis of Nielson and Nielson [4].

Gomard and Jones [6] use a statically-typed two-level language for partial evaluation of the untyped λ-calculus. This language is the basis for many binding-time analyses.

Glück and Jørgensen study partial evaluation in the generalized context where inputs can arrive at an arbitrary number of times rather than just two [5], and demonstrate that binding-time analysis in a multi-level setting can be done with efficiency comparable to that of two-level binding time analysis.

Davies extends the Curry-Howard isomorphism to a relation between temporal logic and the type system for a multi-level language [3].

Moggi [7] advocates a categorical approach to two-level languages based on indexed categories, and stresses formal analogies with a categorical account of phase distinction and module languages.

Acknowledgments: *We would like to thank Bruno Barbier, Jeff Lewis, Emir Pasalic, Yannis Smaragdakis, Eelco Visser, Phil Wadler and Lisa Walton for comments on a draft of the paper.*

References

1. Zine El-Abidine Benaissa, Eugenio Moggi, Walid Taha, and Tim Sheard. A categorical analysis of multi-level languages (extended abstract). Technical Report CSE-98-018, Department of Computer Science, Oregon Graduate Institute, December 1998. Available from `ftp://cse.ogi.edu/pub/tech-reports/README.html`.
2. Olivier Danvy. Type-directed partial evaluation. In *23rd Annual ACM Symposium on Principles of Programming Languages (POPL'96)*, St.Petersburg Beach, Florida, January 1996.
3. Rowan Davies. A temporal-logic approach to binding-time analysis. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science*, pages 184–195, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.
4. Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *23rd Annual ACM Symposium on Principles of Programming Languages (POPL'96)*, St.Petersburg Beach, Florida, January 1996.
5. Robert Glück and Jesper Jørgensen. An automatic program generator for multi-level specialization. *Lisp and Symbolic Computation*, 10(2):113–158, 1997.
6. Carsten K. Gomard and Neil D. Jones. A partial evaluator for the untyped lambda calculus. *Journal of Functional Programming*, 1(1):21–69, January 1991.
7. Eugenio Moggi. A categorical account of two-level languages. In *MFPS 1997*, 1997.
8. Eugenio Moggi, Walid Taha, Zine El-Abidine Benaissa, and Tim Sheard. An idealized MetaML: Simpler, and more expressive (includes proofs). Technical Report CSE-98-017, Department of Computer Science, Oregon Graduate Institute, October 1998. Available from `ftp://cse.ogi.edu/pub/tech-reports/README.html`.
9. Flemming Nielson and Hanne Rijs Nielson. *Two-Level Functional Languages*. Number 34 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.
10. Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. Multi-stage programming: Axiomatization and type-safety. In *25th International Colloquium on Automata, Languages, and Programming*, Aalborg, Denmark, 13–17 July 1998.
11. Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the ACM-SIGPLAN Symposium on Partial Evaluation and semantic based program manipulations (PEPM'97), Amsterdam*, pages 203–217. ACM, 1997.
12. Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. Technical Report CSE-99-007, Department of Computer Science, Oregon Graduate Institute, January 1999. Available from `ftp://cse.ogi.edu/pub/tech-reports/README.html`.
13. Philip Wickline, Peter Lee, and Frank Pfenning. Run-time code generation and Modal-ML. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 224–235, Montreal, Canada, 17–19 June 1998.

Type-Based Decompilation*
(or Program Reconstruction via Type Reconstruction)

Alan Mycroft

Computer Laboratory, Cambridge University
New Museums Site, Pembroke Street, Cambridge CB2 3QG, UK
http://www.cl.cam.ac.uk/users/am

Abstract. We describe a system which decompiles (reverse engineers) C programs from target machine code by type-inference techniques. This extends recent trends in the converse process of compiling high-level languages whereby type information is preserved during compilation. The algorithms remain independent of the particular architecture by virtue of treating target instructions as register-transfer specifications. Target code expressed in such RTL form is then transformed into SSA form (undoing register colouring etc.); this then generates a set of type constraints. Iteration and recursion over data-structures causes synthesis of appropriate recursive C `struct`s; this is triggered by and resolves occurs-check constraint violation. Other constraint violations are resolved by C's casts and `union`s. In the limit we use heuristics to select between equally suitable C code—a good GUI would clearly facilitate its professional use.

1 Introduction

Over the last forty years there has been much work on the compilation of higher-level languages into lower-level languages. Traditionally such lower-level languages were machine code for various processors, but there has been growing widening of the concept of compilation on one hand to permit the lower-level language to be a language like C (often viewed as a 'universal assembler language') and on the other to accompany the translation of terms by a corresponding translation of types—good exemplars are many internal phases of the Glasgow Haskell Compiler [4] which is taken to its logical conclusion in Morrisett et al.'s [8] introduction of 'Typed Assembly Language'. A related strand is Necula and Lee's [9] compiler for proof-carrying code in which user types (including a richer set of types containing value- or range-specification) and compiler-generated types or invariants accompany target code ('proof-carrying code') to enable code to be safely used within a security domain.

Two points which can be emphasised are:

- preserving type information increases the reliability of a compiler by allowing it (or subsequent passes) often to report on internal inconsistency if an invalid transformation occurs instead of merely generating buggy code; and

* A preliminary form of this work was presented at the APPSEM'98 workshop in Pisa.

S.D. Swierstra (Ed.): ESOP/ETAPS'99, LNCS 1576, pp. 208–223, 1999.
© Springer-Verlag Berlin Heidelberg 1999

– compilers are in general many-to-one mappings in which the target code is selected from various equivalent target-code sequences by some notion of efficiency—the more optimising a compiler, in general, the greater the number of source-code phrases that map to a given (generable) target-code sequence.

We consider the use of types and type-inference for the reverse process of decompilation, often called reverse engineering. For the purposes of this paper we take the higher-level code to be C and the lower-level code to be register transfer language (RTL).[1] RTL can be used to express various machine codes in architecture independent manner, but in examples we often use a generic RISC-like instruction set. Another important application which drove this work was a large quantity of BCPL [10] legacy code. BCPL was an untyped fore-runner of C popular at Cambridge for low-level implementation until its replacement with ANSI C around 10 years ago. Being untyped it has a single notion of *vector* which conflates the notions of array and record types in the same way that assembler code does. BCPL is easily translatable to RTL code (and indeed source names can be preserved within RTL as annotations) but the challenge was to invent appropriate structure or array types for data-structures just represented by pointers to vectors.

One might wonder where the RTL code comes from. It can be obtained by simple disassembly (and macro-expansion of instructions to RTL form) of code from assembler files, from object files, directly from compiler output or even from DLL's. Note that currently we assume that code is reasonably identified from data and in particular the current system presumes that procedure boundaries (and even—but less critically—procedure names) are available.

Now we turn to one of the central issues of decompilation—that compilation is a many-to-one map means that we must choose between various plausible alternative high-level representations of given RTL code. This is instantly obvious for the names of local variables which are in general lost in compilation and need to be regenerated; although in general we can only give these rather boring names, we can also recover information from a relocation or symbol table (e.g. in a ELF executable) or from a GUI-driven database to aid serious redevelopment of legacy code. However, there are more serious issues. Identifying loops in a reducible flowgraph is fairly easy but since a good compiler will often translate a "`while (`e`)` C" loop to a loop of the form

$$\texttt{if (}e\texttt{) \{ do } C \texttt{ while (}e\texttt{); \}}$$

we must be prepared to select between or offer the user a choice between various alternatives much like names above.

Note that we do not expect to have types [8] or assertions [9] in the machine code (but if we do these may significantly aid decompilation—it seems unfortunate if aids to program reliability and security make the code-breakers task easier too!). See section 7.

[1] Note the notion of source- and target-language is slightly tangled for a decompiler and so we will stick to C and RTL for concreteness.

We briefly justify decompilation. Apart from the obvious nefarious uses, there are real desires (e.g. in the telecoms industry) to continue to exploit legacy code with guaranteed equivalence to its previous behaviour. Additionally, we expect this project to cast further light on the uses of types of various forms in compilation, including proof-carrying code.

Apart from the glib statement that we decompile RTL to C, certain things do need to be made more precise. We will assume that the RTL has 8-, 16-, 32- and (possibly) 64-bit memory accesses and additionally a push-down stack for allocating temporaries and locals via `push` and `pop`. Moreover, the generated C will assume that `char`, `short`, `int` and `long` will represent these types. `unsigned` can be used as a qualifier as demanded by the code (e.g. triggered by unsigned division, shift or comparison, or by user GUI interaction) but otherwise `signed` forms of these types are generated. Pointers are currently assumed to be 32-bit values and again can only be distinguished from `int` values by their uses. We will see type-inference as the central driver of this process.

This work represents a position intermediate between traditional reverse engineering viewpoints. On one hand, there is decompilation work which has mainly considered control restructuring and tended to leave variables as `int`, to be typecast on use as necessary (Cifuentes [1] is a good example). On the other hand, the formal methods community has tended to see reverse engineering as reconstructing invariants and specifications (e.g. by Hoare- or Dijkstra-style weakest precondition or strongest postcondition techniques—see for example Gannod and Cheng [5]) from legacy code so that it may be further manipulated. It is claimed that a type-based approach can be used for gross-level structuring automatically (possibly with a GUI driver for major choice resolution) whereas exact formal methods techniques are more limited in the size of acceptable problem (e.g. due to the need to prove theorems).

2 Intuitive example

Consider the following straight-line code

```
f:      ld.w  4[r0],r0
        mul   r0,r0,r0
        xor   r0,r1,r0
        ret
```

and a procedure calling standard which uses `r`i as argument and result registers. It is apparent that `f` has (at least) two arguments—see later—but for now we assume exactly two arguments. It is clear that `f` could be expressed as

```
int f(int r0, int r1)
{    r0 = *(int *)(r0+4);
     r0 = r0 * r0;
     r0 = r1 ^ r0;
     return r0;
}
```

However, if we break register uses into *live ranges* and give each a separate name we get:

```
int f(int r0, int r1)
{    int r0a = *(int *)(r0+4);
     int r0b = r0a * r0a;
     int r0c = r1 ^ r0b;
     return r0c;
}
```

Now it is apparent here that argument `r0` could be written as type (`int *`) instead of (`int`) which allows `*(int *)(r0+4)` to be replaced by `*(r0+1)` or its syntactically equivalent form `r0[1]`.[2] Moreover (modulo taking care not to violate any of C's rules concerning side-effects and *sequence points*), variables only used once can be folded into their referencing expressions yielding

```
int f(int *r0, int r1)
{    int r0a = r0[1];
     return r1 ^ (r0a * r0a);
}
```

There is now a further issue of stylistic choice as to whether the above code is preferred or the alternative:

```
int f(int *r0, int r1);
{    return r1 ^ (r0[1] * r0[1]);
}
```

which simply may have generated the original code as a result of a compiler's common sub-expression phase.

We recall the discussion in the introduction in which we observed that the more optimising a compiler the more pieces of code are mapped into a given, possibly optimal, form. A good correctness-preserving heuristic will select one (hopefully readable) form (a maximum-valued solution to various rules). A GUI user interface could select between wide-scale revision (i.e. seeking alternative local—to the constraint solver—maximum) or by demanding a choice between syntactic forms on a local—to the generated source code—basis.

3 SSA—Single Static Assignment

The Single Static Assignment (SSA) form (see e.g. [2]) is a compilation technique to enable repeated assignments to the same variable (in flowgraph-style code) to be replaced by code in which each variable occurs (statically) as a destination exactly once. We use the same technique for decompilation because we wish

[2] Note the possibility that `r0` could be given a type (`struct { int m0, m4, m8; }`) which would then lead to `int r0a = r0->m4;`. There is a notion of polymorphism here and we return to this point later.

to undo register-colouring optimisations whereby objects of various types, but having disjoint lifetimes, are mapped onto a single register.

In straight-line code the transformation to SSA is straightforward, each variable v is replaced by a numbered instance v_i of v. When an update to v occurs this index is incremented. This results in code like

```
v = 3; v = v+1; v = v+w; w = v*2;
```

(with next available index 4 for `w` and 7 for `v`) being mapped to

$$\texttt{v}_7 \texttt{ = 3; } \texttt{v}_8 \texttt{ = v}_7\texttt{+1; } \texttt{v}_9 \texttt{ = v}_8\texttt{+w}_3\texttt{; } \texttt{w}_4 \texttt{ = v}_9\texttt{*2;}$$

On path-merge in the flowgraph we have to ensure instances of such variables continue to cause the same data-flow as previously. This is achieved by placing a logical (single static) assignment to a new common variable on the path-merge node, which captures the effect of two separate assignments on the arcs leading to the path-merge node. This is conventionally represented by a so-called ϕ-function at entry to the path-merge node. The intent is that $\phi(x, y)$ takes value x if control arrived from the left arc or y if it arrived from the right arc; the value of the ϕ-function is used to define a new singly-assigned variable. Thus consider

```
if (p) { v = v+1; v = v+w; } else v=v-1;
w = v*2;
```

which would map to (only annotating `v` and starting at 4)

$$\texttt{if (p) \{ v}_4 \texttt{ = v}_3\texttt{+1; v}_5 \texttt{ = v}_4\texttt{+w; \} else v}_6\texttt{=v}_3\texttt{-1;}$$
$$\texttt{v}_7 \texttt{ = } \phi(\texttt{v}_5,\texttt{v}_6)\texttt{; w = v}_7\texttt{*2;}$$

In examples our variable names will be based on those of machine registers `r0`, `r1`, etc.—instances of these will be given an alphabetic suffix, thus `r0a`, `r4e`, etc.

4 Type reconstruction

Our type reconstruction algorithm is based on that of Milner's algorithm W [7] for ML; it shares the use of unification but involves a rather more complicated type system and delays unification until all constraints are available. Unification failure is used to trigger reconstruction of C types in a way which enables the constraint resolution failure to be repaired.

The C algebra of types does not neatly express the type concepts needed during type reconstruction[3] so we use an internal type algebra (for types t, `struct` members s and register types r) given by:

$$t ::= \mathit{char} \mid \mathit{short} \mid \mathit{int} \mid \mathit{ptr}(t) \mid \mathit{array}(t) \mid \mathit{mem}(s) \mid \mathit{union}(t_1, \ldots, t_k)$$
$$s ::= n_1 : t_1, \ldots, n_k : t_k$$
$$r ::= \mathit{int} \mid \mathit{ptr}(t)$$

[3] Indeed sometimes user-interaction may be desirable to select between C alternatives.

where the n_i range over natural numbers and $k > 0$. α and β are respectively also used to range over t and r (to highlight 'new' type variables generated during unification). The notation $mem(s)$ represents a storage type known to contain various types at identified offsets (i.e. it represents C's `structs` and `unions` and, as we will see, may also represent arrays only accessed by constant subscripts). While the above type grammar allows user interaction to select all C types, for automatic inference it is convenient to require that all $ptr(t)$ types are of the form $ptr(mem(s))$ e.g. by selecting $s = 0 : t$. However, we will still feel free to write (e.g.) $ptr(int)$ to be understood as shorthand. Finally, for the purposes of this paper, $union(t_1, \ldots, t_k)$ is not used as it can be simulated by $mem(0 : t_1, \ldots, 0 : t_k)$.

Each machine code instruction now generates constraints on the types of its operands in a straightforward manner. For example, adopting the notation that tk is the type ascribed to register `rk`, we have

instruction		generated constraint
`mov`	`r4,r6`	$t6 = t4$
`ld.w`	`n[r3],r5`	$t3 = ptr(mem(n : t5))$
`xor`	`r2a,r1b,r1c`	$t2a = int, t1b = int, t1c = int$
`add`	`r2a,r1b,r1c`	$t2a = ptr(\alpha), t1b = int, t1c = ptr(\alpha) \vee$
		$t2a = int, t1b = ptr(\alpha'), t1c = ptr(\alpha') \vee$
		$t2a = int, t1b = int, t1c = int$
`ld.w`	`(r5)[r0],r3`	$t0 = ptr(array(t3)), t5 = int \vee$
		$t0 = int, t5 = ptr(array(t3))$
`mov`	`#42,r7`	$t7 =$ `int`
`mov`	`#0,r7`	$t7 =$ `int` $\vee\ t7 = ptr(\alpha'')$

Note that overloaded C operators such as `+` naturally lead to disjunctive type constraints—compare `add` with `xor`. This also applies to indexed load and store instructions[4] and the constant zero, which is conventionally used to implement the null pointer constant.

Type unification is deferred until section 5, but for now it suffices to consider it as Herbrand unification where occurs-check failures are repaired rather than causing premature termination.

4.1 Inventing recursive data-types from loops or recursion

Consider the C recursive data type

```
struct A { int hd; struct A *tl; };
```

and the iterative and recursive procedures for summing its elements given in Figs. 1 and 2. (Note that for convenience the assembler code is given as a compiler might produce, with the original C code as comment and with generated label names, but note that code and type reconstruction only depends on the machine instructions.) Figs. 3 and 4 show the example assembler code in

[4] Here we assume such instructions do no automatic scaling.

```
;   int f(struct A *x)
;   {   int r = 0;
;       for (; x!=0; x = x->tl) r += x->hd;
;       return r;
;   }
;
f:
        mov     #0,r1
        cmp     #0,r0
        beq     L4F2
L3F2:
        ld.w    0[r0],r2
        add     r2,r1,r1
        ld.w    4[r0],r0
        cmp     #0,r0
        bne     L3F2
L4F2:
        mov     r1,r0
        ret
```

Fig. 1. Iterative summation of a list

```
;   int g(struct A *x)
;   {   return x==0 ? 0 : x->hd + g(x->tl);
;   }
;
g:
        push    r8
        mov     r0,r8
        cmp     #0,r8
        bne     L4F3
        mov     #0,r0
        br      L8F3
L4F3:
        ld.w    4[r8],r0
        jsr     g
        ld.w    0[r8],r1
        add     r1,r0,r0
L8F3:
        pop     r8
        ret
```

Fig. 2. Recursive summation of a list

SSA form and with generated type constraints. We now turn to the process of resolving the type constraints for `f`.

`f:`		$tf = t0 \rightarrow t99$
	`mov r0,r0a`	$t0 = t0a$
	`mov #0,r1a`	$t1a = int \vee t1a = ptr(\alpha_1)$
	`cmp #0,r0a`	$t0a = int \vee t0a = ptr(\alpha_2)$
	`beq L4F2`	
`L3F2:`	`mov` ϕ`(r0a,r0c),r0b`	$t0b = t0a, t0b = t0c$
	`mov` ϕ`(r1a,r1c),r1b`	$t1b = t1a, t1b = t1c$
	`ld.w 0[r0b],r2a`	$t0b = ptr(mem(0 : t2a))$
	`add r2a,r1b,r1c`	$t2a = ptr(\alpha_3), t1b = int, t1c = ptr(\alpha_3) \vee$
		$t2a = int, t1b = ptr(\alpha_4), t1c = ptr(\alpha_4) \vee$
		$t2a = int, t1b = int, t1c = int$
	`ld.w 4[r0b],r0c`	$t0b = ptr(mem(4 : t0c))$
	`cmp #0,r0c`	$t0c = int \vee t0c = ptr(\alpha_5)$
	`bne L3F2`	
`L4F2:`	`mov` ϕ`(r1a,r1c),r1d`	$t1d = t1a, t1d = t1c$
	`mov r1d,r0d`	$t0d = t1d$
	`ret`	$t99 = t0d$

Fig. 3. Iterative sum in SSA form with generated type constraints

Type reconstruction (for `f` using the constraints in Fig. 3) now proceeds by:

Occurs-check constraint failure:

$$t0c = t0b = ptr(mem(4 : t0c)) = ptr(mem(0 : t2a)$$

Breaking cycle with:

`struct G` $\{$ $t2a$ `m0;` $t0c$ `m4; ...`$\}$ i.e.
$t0c = ptr(mem(0 : t2a, 4 : t0c)) = ptr($`struct G`$)$

A record is kept that this particular *mem* is represented by `struct G` which can then be used for printing types. Solving gives two solutions:

$$t0 = t0a = t0b = t0c = ptr(\texttt{struct G})$$
$$t99 = t1a = t1b = t1c = t1d = t2a = t0d = \texttt{int}$$
$$tf = ptr(\texttt{struct G}) \rightarrow \texttt{int}$$

and

$$t0 = t0a = t0b = t0c = ptr(\texttt{struct G})$$
$$t2a = \texttt{int}$$
$$t99 = t1a = t1b = t1c = t1d = t0d = ptr(\alpha_4)$$
$$tf = ptr(\texttt{struct G}) \rightarrow ptr(\alpha_4)$$

`g:`		$tg = t0 \to t99$
	`mov r0,r0a`	$t0 = t0a$
	`push r8`	
	`mov r0a,r8a`	$t8a = t0a$
	`cmp #0,r8a`	$t8a = int \vee t8a = ptr(\alpha_2)$
	`bne L4F3`	
	`mov #0,r0a`	$t0a = int \vee t0a = ptr(\alpha_1)$
	`br L8F3`	
`L4F3:`	`ld.w 4[r8a],r0b`	$t8a = ptr(mem(4 : t0b))$
	`jsr g`	$t0 = t0b, t0c = t99$
	`ld.w 0[r8a],r1a`	$t8a = ptr(mem(0 : t1a))$
	`add r1a,r0c,r0d`	$t1a = ptr(\alpha_3), t0c = int, t0d = ptr(\alpha_3) \vee$
		$t1a = int, t0c = ptr(\alpha_4), t0d = ptr(\alpha_4) \vee$
		$t1a = int, t0c = int, t0d = int$
`L8F3:`	`mov` ϕ`(r0a,r0d),r0e`	$t0e = t0a, t0e = t0d$
	`pop r8`	
	`ret`	$t99 = t0e$

Fig. 4. Recursive sum in SSA form with generated type constraints

The second solution is a parasitic solution which is caused by the effective overloading of addition and the constant zero on both pointers and integers as discussed earlier. It corresponds to creating the variable `r` in the original code as

```
char *r = 0;
```

and then adding on the (`int`) elements `x->hd` by address arithmetic. We believe that this false solution (it corresponds to code which is not strictly ANSI conformant) can be eliminated by enhancing the type system with a *weak pointer* type which is not suitable for arithmetic (cf. `void *` in C); however this awaits experiment.

Having obtained, then, the solution $tf = ptr($`struct G`$) \to$ `int` with

$$\texttt{struct G \{ } t2a \texttt{ m0; } t0c \texttt{ m4; ...\}}$$

we can set about mapping the assembly code into appropriate C. Note that no information has been derived about the size of `struct G`; the use of the ellipsis above corresponds to the type "record type unknown apart from having field `m`" obtained for a Standard ML function such as

```
fun f(x) = x.m;
```

We model this in concrete C by creating an optional padding type `Tpad`.[5] It is now simple to translate the above code into the following C by re-constituting

[5] Unfortunately for us, C does not allow zero-sized types and so we must allow the field to be optional or allow the C pre-processor to macro-expand away `Tpad` if later information (e.g. from uses of the function `f`) indicate its size to be zero.

expressions from variables only used once and by pattern matching (out of the scope of this paper) for commands to obtain:

```
struct G { int m0; struct G *m4; Tpad m8; };
int f(struct G *x)
{   int r = 0;
    if (x != 0)
       do { r += x->m0; x = x->m4; } while (x != 0)
    return r;
}
```

Further pattern matching can reproduce the original `for` loop.

Incidentally, note that the recursive list summation function `g` results in an equivalent set of constraints and therefore can be similarly decompiled into:

```
int g(struct G *x)
{   int r;
    if (x==0)
        r = 0;
    else
    {   int t = g(x->m4);
        r = t + x->m0;
    }
    return r;
}
```

But why is this not nearly so close to the original (even if it is one of the common coding styles for this type of recursion)? Consider the expression

```
x->hd + g(x->tl)
```

which ANSI C declare to be implementation defined if `g()` side-effects `x->hd` and otherwise allows the compiler to choose whether to evaluate `x->hd` or the call to `g` first—sensible compilers would generally evaluate `g(x->tl)` first since this reduces register pressure. However, conversely, we are not in general at liberty to fold a *sequential* call to `g()` and an addition of `x->hd` into the original code and hence the above decompilation is as good as we can obtain under the assumption that procedure calls (e.g. `jsr g`) can affect memory arbitrarily. However, *if* we could determine that the call `jsr g` cannot result in side-effects (on `x->hd`) then the following simplifications are triggered:

- the code for the `else`-part could be reconstructed to

  ```
  r = x->m0 + g(x->m4);
  ```

  which is only valid C if `g()` cannot affect `x->m0`
- then, given that both consequents assign to `r`, the whole body simplifies to the original

  ```
  return x==0 ? 0 : x->hd + g(x->tl);
  ```

This short-coming of the present type-based approach could be remedied by extending function types to include details of side effects—using a *type and effect system* (also known as an *annotated type system*)—see for example [11]

Finally, we observe that all the suggested decompilations of `f` and `g` above yield identical target code when processed by the compiler (ncc) which was used to produce the sample code for decompilation. Of course, we might be able to use a better compiler the second time round!

4.2 When `structs` cannot resolve type conflicts

Although our decompiler needs internally a richer set of types than ML (e.g. we have seen that `ld.w 4[r0],r1` leads us to reason that `r0` may be a pointer to any type with a 32-bit component at offset 4, including both structures and arrays) we have exploited constraint gathering and solution by unification much as we might find in an ML compiler. In section 5 we will discuss the additional ordering on types (and non-Herbrand unification) occasioned by code which can reflect either array element or struct member access.

(Herbrand) unification may fail for two reasons. Firstly, a type variable may need to be unified with a term containing it—this is solved as above by synthesising recursive data types. Secondly, we may have a straightforward clash of type-constructors and it is to this case which we now turn.

Consider the code:

```
h:      ld.w    4[r0],r1
        xor     r1,r0,r0
        ret
```

where `r0` is constrained to be an `int` because of its appearance as the source of an `xor` instruction and as a pointer to store (containing an `int` at offset 4) due to the `ld.w` instruction. (Note that all the uses of `r0` except for the destination of the `xor` instruction form a single live range and so the transformation to SSA form used in the introduction does not help here.) So we attempt to unify `int` with $ptr(mem(4 : \texttt{int}))$ and find no solution. Such situations are deferred until the global set of constraint failures are available (here there are no more) and then the application to typing outlined by Gandhe et al. [3] for finding maximal consistent subsets of inconsistent sets is applied.

Here we find a benefit of using C as the high-level language for decompilation in that it can express such code by *casts* or `union` types. C's `union` type can express the solution trivially as

```
int h(union {int i; int *p;} x) { return x.p[1] ^ x.i; }
```

but this is not a very common (nor very readable) form of C and indeed is not strictly conforming in ANSI C (reading a `union` at a different type from what it was written is forbidden). We would prefer to restrict the synthesis of `union`s to within generated `struct`s which contain also a discriminator. Cast-based alternatives seem better in this case and we get three plausible solutions:

```
int h1(int x) { return *(int *)(x+4) ^ x; }
int h2(int *x) { return  x[1] ^ (int)x; }
struct h3arg { Tpad1 m0; int m; Tpad2 m8; };
int h3(struct h3arg *x) { return x->m ^ (int)x; }
```

Note we have suppressed the variant of `h1` in which `x` is cast to a new `struct` type which contains an `int` at offset 4; clearly a skilled program re-constructor might be able to specify the `*(int *)(x+4)` more precisely, but inventing a separate new datatype for each such access would clutter code for no clear benefit. We will prefer option `h3` by default (with the understanding that the generated `struct h3arg` will be unified with arguments of callers), leaving array creation to be triggered by non-constant indexing (or user GUI interaction); the next section investigates the choice between arrays and `struct`s in more detail.

Of course, one justification of using C in this paper is that the above assembler code could not plausibly be generated by any Haskell compiler— C is more expressive in this sense.

4.3 Arrays versus `struct`s

The approach we have taken so far has been to use `struct`s whenever possible. While these, together with casts and address arithmetic, would suffice for decompilation, it is more rational to trigger array synthesis when indexing instructions occur, whether they be manifest:

```
ld.w    (r5)[r0],r3
```

or more indirectly coded (a non-constant `int` value being used for addition or subtraction at pointer type) such as

```
add     r5,r0,r1
ld.w    0[r1],r3
```

Such an indexing instruction (for the purposes of this discussion we will assume scaling is not done in hardware, thus the effective address is $(\texttt{r0}) + (\texttt{r5})$) generates constraints (as explained earlier):

$$\texttt{ld.w (r5)[r0],r3}\ t0 = ptr(array(\beta)), t5 = int, t3 = \beta \vee \\ t0 = int, t5 = ptr(array(\beta)), t3 = \beta$$

where β is constrained to be a register type, i.e. `int` or $ptr(\alpha)$.

If the constraints for a given pointed-to type are all `struct` types (resulting from constant offsets) then the resulting unified type is also `struct` as in the previous subsection. Otherwise, if all accesses via a pointer are of the same *size*, e.g. all 32-bit accesses, then the unified type is *array*, otherwise a `union` type is generated, e.g. the constraints for

```
ld.b    0[r0],r1
ld.b    48[r0],r2
ld.w    (r5)[r0],r3
```

unify to yield

```
union G { struct { char m0; char pad1[47]; char m48; } u1;
          int u2[];
        } *r0;
```

Inferring limits for arrays requires, in general, techniques beyond those available to our type-based reconstruction. If presented with proof-carrying code [9] then array bounds could be extracted from code proved to be safe. To a large extent however, C programmers do not take great care with array size specifications, especially when passed as arguments since the C standard requires formal array parameters to be mapped to pointers thereby losing size information.

Although currently not implemented, note that a GUI could be used to direct that the above `union` should instead be decompiled as

```
struct G { char m0; char pad1[3]; int m4[15]; char m48; } *r0;
```

when it is clear to a user that the array is actually part of the struct. We return to this point in section 6.1.

5 Type Unification

Unification of our types is Herbrand-based with the following additional rules, i.e. the cases below are tried in order if Herbrand unification fails.

- type variable α unifies with type t containing α to yield $t[\texttt{struct G}/\alpha]$ with an auxiliary definition of `struct G` being produced.
- $array(t)$ and $mem(n_1 : t_1, \ldots n_k : t_k)$ unify to $array(t)$ when type $(\forall i) t_i = t$.
- $mem(s_1)$ and $mem(s_2)$ unify to $mem(s_1 \cup s_2)$; note that keeping conflicting items (e.g. $mem(0 : int, 1 : char)$) is not an error since this may later be used to replace the member at offset zero with a `union` in the generated C.

6 Selecting C types for generated types

As noted earlier, generated types are more expressive than C types, and this sometimes means that a choice has to be made from among various possibilities. Moreover certain C types are less commonly used that others, e.g. a function parameter is more likely to be described as (`int *`) rather than (`int (*)[10]`), whereas appropriate uses would leave to identical target code. The default method for selecting types which result from unification is as follows:

- translate *char*, *short*, *int* as `char`, `short`, `int`;
- translate $ptr(t)$ to `T *` where `T` translates t;
- translate $array(t)$ to `T []` where `T` translates t;
- translate $mem(s)$ to:
  - `struct G` if `struct G` was generated during unification for this mem;
  - `T` if $s = (0 : t)$ and `T` translates t;
  - `struct G` where `G` is a new struct definition laying out translated typed members of s at appropriate offsets (note this may require `union`s for overlapping members of s and may require padding members for unreferenced struct elements).

6.1 Unwelcome choice in reconstructing arrays and structs

The main problem which arises is due to the expressiveness (and defined storage layout) of C's `struct`, `union` and array types compared to those of Java (which may only contain another such object via a pointer). As in Fortran it can be hard to distinguish array type `int [10][10]` from `int[100]`. Similarly, arrays or structs containing other arrays or structs cannot in general be uniquely decoded, consider distinguishing objects `x1`, ... `x3` defined by:

```
struct S1 { int a; int b[4]; int c; int d[4]; } x1;
struct S2 { int a; int b[4]; } x2[2];
struct S3 { struct S2 c,d; } x3;
```

We are exploring various options for constraint resolution when array indexing and `struct` selection occurs. Consider code like that discussed in section 4.3:

```
ld.b    0[r0],r1
ld.b    48[r0],r2
ld.w    (r5)[r0],r3
```

There are several possible ways to approximate this data-structure from the above information, including:

```
union T1 { char a[/*ATLEAST*/49]; int b[/*ATLEAST*/17]} *r0;
struct T2 { char a; char pad[3]; int b[15]; char c; } *r0;
```

The latter is appealing in that additional information, e.g. a

```
ld.b    16[r0],r4
```

instruction could cause natural, fuller-information, revision to

```
struct T2 { char a; char pad[3]; int b[3]; } (*r0)[4];
```

Finally, a current limitation is not exploiting stride information. For example, we could use information about the computation of `r0` to determine restrictions on sizes (and hence types) in the pointed-to values represented by `r5` in instructions like `ld.w  (r5)[r0],r3`.

7 Conclusions and further work

We have described a system which can decompile assembler code at the RTL level to C. It can successfully create `structs` both intra- and inter-procedurally and in doing can generate code close to natural source form.

We have not discussed the use of local variables stored on the stack rather than in registers. A simple extension can manipulate local stacks satisfactorily (essentially at the representative power of Morrisett et al.'s [8] Typed Assembly Language) when local variables are not address-taken. However, there are problems with taking addresses of local stack objects in that it can be unclear

as to where the address-taken object ends—a `struct` of size 8 bytes followed by a coincidentally contiguously allocated `int` can be hard to distinguish from a `struct` of size 12 bytes.

Here it is worth remarking on the assistance given by proof-carrying code [9], particularly when the proof has been generated by a compiler, to our decompiler. Many of the questions which gave difficulty for decompilation concerned issues like: where arrays live inside a `struct`, is the data-structure really an array of `struct`s instead, or simply where a given array (determined by variable indexing) begins and ends. In general these are exactly the points which an accompanying proof-of-safety must address. This suggests that we can probably do much better at decompilation given the proof part—perhaps this shows that proof-carrying code is not a good idea for secret (or deliberately obfuscated) algorithms!

Note that the decompilation process does not depend intrinsically on C. We chose C because of its ability to capture most sequences of machine instructions naturally; casts can also represent type cheating in reconstructed source form. It also provides a good balance of problem-statement and tractability for this initial work. Of course, there are instruction sequences which are not translatable to C—the most obvious example is that C does not have label variables and so `jmp r0` cannot be decompiled (except possibly as a tail-recursive call to a procedure variable).

One could imagine a generalisation of this system where compiler translation rules (e.g. for Haskell) are made available to the decompiler to reconstruct rather more high-level languages. Failure of code to match such rules would in general indicate a call to a native (in the Java sense) procedure or that the proffered code cannot be expressed in the source code represented by the translation rules. This clearly links to the "formal methods" view of reverse engineering discussed in the introduction for inventing higher-level (than C) notions for existing code. Our type-based approach clearly can assist in this process in that it is coarser-grain than algebraic semantic approaches yet retains aspects of global understanding. Work in progress concerns identification and replacement of abstract data-types—in BCPL (a fore-runner of C) adjacent words in memory are required to have addresses differing by one which causes current translators for byte-addressed targets (whether via C or direct to target machine code) to generate large numbers of shift-left or shift-right by two instructions. Identifying "BCPLaddress" as an ADT and re-implementing it could eliminate all such shifts with the exception of those caused by users explicitly relying on this part of the standard.

Finally, we turn to performance: we as yet have no experimental results[6] for large bodies of code, but the ability to reconstruct datatypes for both iterative and recursive procedures is appealing over other techniques. Since the process of data-structure reconstruction depends only on finding cycles which in reality are likely to be quite short even in large programs, we are optimistic about the scalability of the techniques. In common with several type-based systems, interprocedural versions seem to come naturally and without great cost.

[6] A student project is currently underway.

Acknowledgments

I would like to thank the anonymous referees for their helpful comments on the draft version of this paper. Thanks are also due to Pete Glasscock (a student on the Diploma in Computer Science at Cambridge in 1997–98) for illustrating, in his dissertation [6], what could be done in the way of decompilation without type-based reconstruction of source.

References

1. Cifuentes, C. *Reverse Compilation Techniques*, PhD thesis, Queensland University of Technology, 1994. Available as `ftp://ftp.csee.uq.edu.au/pub/CSM/dcc/decompilation_thesis.ps.gz`
2. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N. and Zadeck, F.W. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451-490, October 1991.
3. Gandhe, M., Venkatesh, G., Sanyal, A. Correcting Errors in the Curry System. In Chandrum V. and Vinay, V. (Eds.): Proc. of 16th conf. on Foundations of Software Technology and Theoretical Computer Science, LNCS vol. 1180, Springer-Verlag, 1996.
4. Glasgow Haskell Compiler.
5. Gannod, G.C. and Cheng, B.H.C. Using Informal and Formal Techniques for the Reverse Engineering of C Programs. Proc. IEEE International Conference on Software Maintenance, 1996.
6. Glasscock, P.E. An 80x86 to C Reverse Compiler. Diploma in Computer Science Dissertation, Computer Laboratory, Cambridge University, 1998.
7. Milner, R. A Theory of Polymorphism in Programming, *JCSS* 1978.
8. Morrisett, G., Walker, D., Crary, K. and Glew, N. From System F to Typed Assembly Language. Proc. 25th ACM symp. on Principles of Programming Languages, 1998.
9. Necula, G.C. and Lee, P. The Design and Implementation of a Certifying Compiler. Proc. ACM conf. on Programming Language Design and Implementation, 1998.
10. Richards, M. and Whitby-Strevens, C. BCPL—The Language and its Compiler, *CUP* 1979.
11. Tang, Y.M., Jouvelot, P. Effect Systems with Subtyping. Proc. ACM symp. on Partial Evaluation and Program Manipulation (PEPM), 1995.

An Operational Investigation of the CPS Hierarchy

Olivier Danvy[1] and Zhe Yang[2] ⋆

[1] BRICS ⋆⋆⋆
Department of Computer Science, University of Aarhus
Building 540, Ny Munkegade, DK-8000 Aarhus C, Denmark
danvy@brics.dk
[2] Department of Computer Science, New York University
251 Mercer Street, New York, NY 10012, USA
zheyang@cs.nyu.edu

Abstract. We explore the hierarchy of control induced by successive transformations into continuation-passing style (CPS) in the presence of "control delimiters" and "composable continuations". Specifically, we investigate the structural operational semantics associated with the CPS hierarchy.
To this end, we characterize an operational notion of continuation semantics. We relate it to the traditional CPS transformation and we use it to account for the control operator **shift** and the control delimiter **reset** operationally. We then transcribe the resulting continuation semantics in ML, thus obtaining a native and modular implementation of the entire hierarchy. We illustrate it with several examples, the most significant of which is layered monads.

1 Introduction

1.1 Background

Continuation-passing style (CPS) programs are usually obtained by CPS transformation. The CPS hierarchy is obtained by iterating the CPS transformation, which yields programs whose types obey the following pattern:

$$\begin{aligned} \mathrm{Fun} &= \mathrm{Val}_0 \to \mathrm{Cont}_1 \to \mathrm{Cont}_2 \to \ldots \to \mathrm{Cont}_n \to \mathrm{Ans}_n \\ \mathrm{Cont}_1 &= \mathrm{Val}_1 \to \mathrm{Cont}_2 \to \ldots \to \mathrm{Cont}_n \to \mathrm{Ans}_n \\ &\ldots \\ \mathrm{Cont}_n &= \mathrm{Val}_n \to \mathrm{Ans}_n \end{aligned}$$

In the CPS hierarchy, programs exhibit the familiar pattern of success/failure continuations which is pervasive in functional specifications of backtracking.

⋆ Partially supported by National Science Foundation grant CCR-9616993 and by BRICS.
⋆⋆⋆ Basic Research in Computer Science (http://www.brics.dk), Centre of the Danish National Research Foundation.

S.D. Swierstra (Ed.): ESOP/ETAPS'99, LNCS 1576, pp. 224–242, 1999.
© Springer-Verlag Berlin Heidelberg 1999

Enriching CPS with identity and composition makes it possible to simulate Prolog-style backtracking and also to accumulate results. To simulate Prolog-style backtracking, we successively apply the current continuation to all possible choices—failing if there are none. To accumulate results, we use the current continuation to compute the result of the "remaining computation" and stash it away in an accumulator. To combine these two control mechanisms, e.g., to accumulate all the possible results of a non-deterministic computation in a list, we exploit the natural hierarchy between these two processes: the generation should take place in a context where its successive results are accumulated. We therefore CPS-transform the generation process (thus making its continuation continuation-passing) and we supply the accumulation process as its initial continuation [5].

The CPS hierarchy thus offers a fitting platform to express hierarchical backtracking—at the price indicated by the types above: a quadratic inflation of continuations.

The last level of CPS can be avoided by using the identity continuation and the ability to compose continuations. This is often deemed enough when $n = 1$ or $n = 2$ in the type equations above. For higher values of n, this quadratic cost can be alleviated with a linguistic device: two new syntactic forms in direct style, whose CPS transformation yields the desired effect of initializing a continuation with identity and of composing continuations. Initializing a continuation with identity is achieved with the control delimiter **reset**. Composing continuations is enabled by the control operator **shift** which captures the current (delimited) continuation and makes it ready to be composed with a subsequent continuation [5,6]. For comparison, the control operator **callcc** captures the current (unlimited) continuation and makes it ready to replace a subsequent continuation [17,23].

The challenge now is how to implement the CPS hierarchy more directly than by repeated CPS transformations and more efficiently than with a definitional interpreter [5]. Filinski showed how to implement the first level natively, using **callcc** and one reference cell [12]. In this article, we show how to implement the entire hierarchy natively, using **callcc** and one reference cell per level.

1.2 Related work

The CPS hierarchy was identified and advocated by Danvy and Filinski [5], who also introduced the corresponding hierarchy of control operators $\mathbf{shift}_n$ and $\mathbf{reset}_n$ (one per surrounding continuation). At the same time, but independently of CPS, Felleisen invented control delimiters [8], initiating a whole area of work on composable continuations and hierarchies of control [9,10,16,18,19,21,22,27], [31,33,34]. Control delimiters, for example, were instrumental to obtain a full-abstraction result [35].

All researchers in this new area followed Felleisen and defined their new control constructs operationally. They reported a variety of control operators, each of these displaying inventiveness in its modus operandi, its description, and its implementation.

In contrast, **shift** and **reset** are defined by translation into CPS. They have also proven particularly fruitful: because (we believe) control delimiters and composable continuations arise naturally in the CPS hierarchy, a number of applications of **shift** and **reset** were reported through the 90's [4,12,14,24,25,37], up to and including the R5RS [23]. There were only two further studies, however, of the CPS hierarchy and its guidelines: Murthy's, formalizing its type system [28], and Filinski's, establishing its equivalence with computational monads [13,15].

1.3 This work

The growing number of applications of **shift** and **reset** leads one to want to combine them. For example, suppose that we want to specialize programs that use **shift** and **reset**, using type-directed partial evaluation [4]. The problem is that type-directed partial evaluation also uses **shift** and **reset**, and we would like these two uses not to interfere with each other. This kind of applications require the CPS hierarchy to layer different uses of **shift** and **reset** at different levels.

It is difficult to implement the CPS hierarchy natively, since the semantics of built-in constructs cannot be altered. We thus take a novel approach using operational semantics: we characterize an operational 'continuation semantics' and following Section 1.1, (1) we enrich it with identity and composition of continuation as provided by **shift** and **reset**, and (2) we transform the result in a new continuation semantics. The new semantics extends the old one with a new pair of **shift** and **reset**; moreover, it can be natively implemented in the old semantics, since all the rules in the new semantics except those of the newly added operators are those of the old semantics, with the addition of one unchanged component. Iterating this process yields a family of semantics—the CPS hierarchy—and its native and modular implementation in ML, à la Filinski [12,13,15]. This general approach provides a native implementation of the new language constructs.

En passant, to make sure that we account for **shift** and **reset** as originally defined (i.e., by CPS transformation), we relate our operational notion of continuation semantics with the traditional CPS transformation.

1.4 Applications

A toy example: The two following computations declare an outer context `1 + [ ]`. They also declare a delimited context `[50 + [ ]]`, which is abstracted as a function denoted by `k`. This function is successively applied to `0`, yielding `50`, and to `10`, yielding `60`. These two results are added, yielding `110` which is then plugged in the outer context. In both cases, the overall result is `111`.

```
1 + reset (fn () => 50 + shift (fn k => (k 0) + (k 10)))

1 + let fun k v = 50 + v in (k 0) + (k 10) end
```

In the first computation, the context is delimited by `reset` and the delimited context is abstracted into a function with `shift`. The second computation is the continuation-passing counterpart of the first one.

More substantial examples: The CPS hierarchy also makes it possible to express computations like min-max processes or quantifier alternation. For example, the existential quantifier $\exists v.p(v)$ for a condition p over non-deterministically generated values v can be implemented as

```
fun exist v p
    = shift (fn k => if (p v) then true else k ())
```

where the return value of **reset** corresponding to the collection is set to `false`. Similarly, we can implement the universal quantifier with a function `forall`. Using these two functions at different levels, we can write formulae of arbitrary quantifier alternation.

2 Operational Semantics of the CPS Hierarchy

This section presents a family of operational semantics that can be directly transcribed into an implementation.

Starting with an operational semantics S for ML (Section 2.1), we characterize an operational notion of continuation semantics (Section 2.2). We then relate continuation semantics and syntactic CPS transformation, which is a result in itself (Sections 2.3 and 2.4). Based on the CPS transformation, we provide a semantic account of **shift** and **reset** (Section 2.5).

The semantics L, which is S extended with **shift** and **reset**, is no longer a continuation semantics. We induce two continuation semantics H and I, and prove that they both simulate the semantics L (Sections 2.6 and 2.7). Moreover, I is directly implementable in S, in that the S-rules embed into the I-rules with the addition of one unchanged component. This component can be implemented by a reference cell. As for the remaining I-rules, they correspond to new control operators which can be implemented as functions.

The resulting semantics I is a continuation semantics, and thus, generalizing, we can iterate the whole transformation (Section 2.8). The resulting family of operational semantics formalizes the CPS hierarchy and is directly implementable in the initial operational semantics of ML (Section 3).

2.1 Starting semantics S

We use Harper, Duba, and MacQueen's "continuation-based operational semantics" for ML [17] as our starting semantics S.[1] Its syntactic categories are defined by the following grammar.

$$\begin{aligned} e \in \mathit{Exp} &::= x \mid \ell \mid \lambda x.e \mid e_0\, e_1 && \text{—expressions} \\ v \in \mathit{Val} &::= \ell \mid \lambda x.e && \text{—values} \\ k \in \mathit{Cont} &::= \Box \mid k\, e \mid v\, k && \text{—continuations} \end{aligned}$$

Its inference rules specify a judgment of the form "$k \vdash e \Rightarrow v$" which reads "under the continuation k, evaluating the expression e yields the answer v" (Table 1).

[1] For brevity, both WRONG and LET rules in the original semantics are omitted here. The WRONG rule specifies the error case and serves in the formulation of type soundness. The LET rule is only there for ML's let polymorphism.

$$(\mathsf{VAL0})\frac{}{\Box \vdash v \Rightarrow v} \qquad (\mathsf{VAL1})\frac{\Box \vdash k[v_1] \Rightarrow v}{k \vdash v_1 \Rightarrow v} \quad (k \neq \Box)$$

$$(\mathsf{FN})\frac{k[\Box\, e_1] \vdash e_0 \Rightarrow v}{k \vdash e_0\, e_1 \Rightarrow v} \ (e_0 \text{ not a value}) \quad (\mathsf{ARG})\frac{k[v_0\, \Box] \vdash e_1 \Rightarrow v}{k \vdash v_0\, e_1 \Rightarrow v} \ (e_1 \text{ not a value})$$

$$(\mathsf{BETA})\frac{k \vdash [v_1/x]e \Rightarrow v}{k \vdash (\lambda x.e)\, v_1 \Rightarrow v}$$

Table 1. An operational continuation semantics

A continuation k can be thought of as an expression with precisely one "hole" $\Box$ in it. We write $k[e]$ and $k[k']$ to denote the expression and continuation obtained by filling the hole in k with an expression e and a continuation k', respectively, where $k[k']$ is the "composition" of k with k'.

We are mainly interested in the dynamic semantics of **shift** and **reset**, and thus we do not present typing rules and the related soundness proof, which can be adapted from the work of Harper, Duba, and MacQueen [17] and of Gunter, Rémy, and Riecke [16]. We rely on the static type system of our implementation language, ML, for the type soundness (Section 3).

2.2 An operational notion of continuation semantics

Operational semantics give rise to derivation trees. We define a *branchless semantics* as an operational semantics whose rules have one premise at most. Such a semantics gives rise to *branchless* derivation trees, i.e., lists. Note that a branchless semantics directly corresponds to a reduction semantics, where reduction proceeds from the conclusion of a rule to the premise, or to the final result if the rule has no premise. Staying in the world of branchless evaluation semantics makes it easy to refer to a complete computation (as a judgment) as well as a single reduction (as a rule instance).

A continuation semantics, like the one in Table 1, is branchless: the continuation component in its judgments keeps track of the remaining branches in a corresponding direct-style derivation tree; it can be regarded as the stack used to traverse this derivation tree.

2.3 CPS transformation

Previous studies of the CPS hierarchy build on the CPS transformation. To justify our study of control operators with the continuation semantics of Table 1, we adapt the call-by-value CPS transformation to its expressions, values, and continuations.

$$\begin{aligned} [\![x]\!]_{Exp} &= \lambda\kappa.\kappa\, x \\ [\![v]\!]_{Exp} &= \lambda\kappa.\kappa\, [\![v]\!]_{Val} \\ [\![e_0\, e_1]\!]_{Exp} &= \lambda\kappa.[\![e_0]\!]_{Exp}\, \lambda v_0.[\![e_1]\!]_{Exp}\, \lambda v_1.v_0\, v_1\, \kappa \end{aligned} \qquad \begin{aligned} [\![\ell]\!]_{Val} &= \ell \\ [\![\lambda x.e]\!]_{Val} &= \lambda x.[\![e]\!]_{Exp} \end{aligned}$$

$$\begin{aligned}
[\![\Box]\!]_{Cont} &= \lambda v.\lambda\kappa.\kappa\, v \\
[\![k\, e_1]\!]_{Cont} &= \lambda v.\lambda\kappa.[\![k]\!]_{Cont}\, v\, \lambda v_0.[\![e_1]\!]_{Exp}\, \lambda v_1.v_0\, v_1\, \kappa \\
[\![v_0\, k]\!]_{Cont} &= \lambda v.\lambda\kappa.[\![k]\!]_{Cont}\, v\, \lambda v_1.[\![v_0]\!]_{Val}\, v_1\, \kappa
\end{aligned}$$

The rationale for $[\![\cdot]\!]_{Cont}$ is that the "hole" $\Box$ in the continuation is an abstraction over a value, as captured in the following lemma, where "$=_{\beta\eta}$" denotes $\beta\eta$-convertibility.

Lemma 1. *For all $k \in Cont$ and $v \in Val$, $[\![k[v]]\!]_{Exp} =_{\beta\eta} [\![k]\!]_{Cont}\, [\![v]\!]_{Val}$.*

Proof. By structural induction on k.

2.4 Soundness and completeness of the operational semantics

The following theorem connects the continuation semantics of Section 2.1 and the CPS transformation of Section 2.3.

Theorem 1. *For all $k \in Cont$, $e \in Exp$, and $v \in Val$, if $k \vdash e \Rightarrow v$ then*

$$[\![e]\!]_{Exp}\, \lambda w.[\![k]\!]_{Cont}\, w\, \lambda a.a =_{\beta\eta} [\![v]\!]_{Val}.$$

Proof. By rule induction on $k \vdash e \Rightarrow v$.

Corollary 1. *For all $e \in Exp$ and $v \in Val$, if $\Box \vdash e \Rightarrow v$ then $[\![e]\!]_{Exp}\, \lambda w.w =_{\beta\eta} [\![v]\!]_{Val}$.*

Because the term $[\![e]\!]_{Exp}\, \lambda w.w$ is convertible to a value $[\![v]\!]_{Val}$, it must reduce to a *value* that is equal to $[\![v]\!]_{Val}$ modulo $\beta\eta$-conversion under normal-order evaluation (by the Normalization theorem), and thus also under applicative-order evaluation (by Plotkin's Indifference theorem [29]). The operational semantics is thus sound with respect to the call-by-value semantics defined by the CPS transformation.

Proving completeness requires a close correspondence between the rules and the translated terms, and as often, "administrative redexes" in the CPS transformation get in the way. To prove completeness (i.e., that the evaluation of the CPS form of a term e with an identity continuation leads to a value v, then $\Box \vdash e \Rightarrow v$), we successfully adopted Danvy and Filinski's one-pass CPS transformation [6].

These soundness and completeness results are not surprising. One can also obtain them by proving the equivalence of the continuation semantics and a direct semantics, and then by using Plotkin's Simulation theorem [29]. A more immediate connection between continuation semantics and CPS transformation, however, provides the basic framework for adding control operators.

2.5 An operational account of shift and reset

A control operator "reifies" a continuation k into a function f_k. In terms of the CPS transformation, such a function appears as a λ-term:

$$\Lambda_k = \lambda w.[\![k]\!]_{Cont}\, w\, \lambda a.a.$$

Correspondingly, in the continuation semantics, we need to find out how to invoke such a function f_k to be able to use a reified continuation.

Let us fix a continuation k. For any given value v, the corresponding v' such that $k \vdash v \Rightarrow v'$ is unique, if it exists. This suggests us to define a function f_k for every continuation k as follows.

$$f_k\, v = v' \iff k \vdash v \Rightarrow v'$$

And this is justified by the CPS transformation: as a corollary of Theorem 1 when $e = v$, the term $\Lambda_k\,([\![v]\!]_{Val})$ is $\beta\eta$-convertible to

$$(\lambda w.[\![k]\!]_{Cont}\, w\, \lambda a.a)\,[\![v]\!]_{Val} =_{\beta\eta} [\![v]\!]_{Exp}\, \lambda w.[\![k]\!]_{Cont}\, w\, \lambda a.a =_{\beta\eta} [\![v']\!]_{Val}.$$

Now we are ready to introduce the control operators **shift** and **reset** by adding the following expression forms and the corresponding rules.

$$e ::= ... \mid \textbf{reset}\ e \mid \textbf{shift}\ c.e \mid \textbf{pushcc}(e, k)$$

Shift and **reset** are defined by their CPS transformation [5,6]:

$$\begin{aligned}[\![\textbf{shift}\ c.e]\!]_{Exp} &= \lambda\kappa.(\lambda c.[\![e]\!]_{Exp}\, \lambda a.a)\, \lambda w.\lambda\kappa'.\kappa'\, (\kappa\, w)\\ &\qquad \text{—composition of continuations}\\ [\![\textbf{reset}\ e]\!]_{Exp} &= \lambda\kappa.\kappa\, ([\![e]\!]_{Exp}\, \lambda a.a)\\ &\qquad \text{—identity continuation}\end{aligned}$$

In both **shift** $c.e$ and **reset** e, the type of e should be the same—though not necessarily the same as the final result type. Thus any **shift**-expression must be delimited by a corresponding **reset** or a **shift**—a hidden restriction that cannot be easily expressed in this translation. We address it in Section 2.6.

The translation of **shift** and **reset** are not in CPS because they compose continuations. This programming pattern is abstracted by the meta-control operator **pushcc**:

$$[\![\textbf{pushcc}(e, k)]\!]_{Exp} = \lambda\kappa.\kappa\, ([\![e]\!]_{Exp}\, \lambda w.[\![k]\!]_{Cont}\, w\, \lambda a.a)$$

Pushcc is a meta-control operator because its expression form contains continuations. It therefore can only be used to define other control operators.

Correspondingly, in the operational semantics, we can express the composition of functions f_k and f'_k related to continuations k and k' by adding the rule pushcc (Table 2). As for **shift** and **reset**, they are defined by the rules shift and reset in term of **pushcc**.

Let us resume the inductive proof of Theorem 1 for the three new rules. We only reproduce the most interesting one here, i.e., pushcc.

Proof. (excerpt)

The induction hypotheses for the rule pushcc (Table 2) read:

$$[\![e]\!]_{Exp}\, \lambda w.[\![k']\!]_{Cont}\, w\, \lambda a.a =_{\beta\eta} [\![v']\!]_{Val} \qquad \text{(i1)}$$

$$[\![v']\!]_{Exp}\, \lambda w.[\![k]\!]_{Cont}\, w\, \lambda a.a =_{\beta\eta} [\![k]\!]_{Cont}\, [\![v']\!]_{Val}\, \lambda a.a =_{\beta\eta} [\![v]\!]_{Val} \qquad \text{(i2)}$$

$$(\mathsf{shift})\frac{\Box \vdash (\lambda c.e)\,\lambda w.\mathbf{pushcc}(w,k) \Rightarrow v}{k \vdash \mathbf{shift}\ c.e \Rightarrow v} \qquad (\mathsf{reset})\frac{k \vdash \mathbf{pushcc}(e,\Box) \Rightarrow v}{k \vdash \mathbf{reset}\ e \Rightarrow v}$$

$$(\mathsf{pushcc})\frac{k' \vdash e \Rightarrow v' \qquad k \vdash v' \Rightarrow v}{k \vdash \mathbf{pushcc}(e,k') \Rightarrow v}$$

Table 2. Definitional rules for **shift** and **reset**

Now, we have $[\![\mathbf{pushcc}(e,k')]\!]_{Exp}\,\lambda w.[\![k]\!]_{Cont}\,w\,\lambda a.a$
$=_{\beta\eta}\ (\lambda\kappa.\kappa\,([\![e]\!]_{Exp}\,\lambda w.[\![k']\!]_{Cont}\,w\,\lambda a.a))\,\lambda w.[\![k]\!]_{Cont}\,w\,\lambda a.a$
$=_{\beta\eta}\ [\![k]\!]_{Cont}\,[\![v']\!]_{Val}\,\lambda a.a$, by (i1)
$=_{\beta\eta}\ [\![v]\!]_{Val}$, by (i2)

Theorem 1, and consequently Corollary 1, still hold for the operational semantics and the CPS transformation extended with the control operators. Therefore, the operational semantics gives the same definition for the control operators as those given by the CPS transformation.

2.6 A continuation semantics for shift and reset

With the addition of pushcc, the semantics is no longer branchless. To obtain an equivalent branchless semantics, we need to use the idea of the CPS transformation, i.e., flattening derivation trees by remembering branching computations (continuations k in pushcc) in a stack.

We thus induce another semantics H from the extended semantics, referred to as L. For clarity, we subscript rules and domains by the semantics they belong to. A judgment in H is of the form $k \vdash_H e \Rightarrow v$, where $e \in Exp_L$ (which can be one of the form introduced by the control operators), but k forms a new domain of "global" continuations: $k \in Cont_H = Cont_L \times (Cont_L\ \mathsf{list}) = Cont_L^+$. The global continuations (of type $Cont_H$) are always non-empty lists of continuations, whose head is the current active continuation, and whose tail is a stack of saved continuations.

The H-rules are given in Table 3. Most of them are simply the corresponding L-rules with the stack ks carried around unchanged. The interesting rules, $\mathsf{pushcc_H}$ and $\mathsf{VAL0_H^{CONS}}$, function as the branching rule $\mathsf{pushcc_L}$.

The semantics H is branchless. We would like to show that it correctly accounts for L, i.e., $\forall\, k, e, v.\ (k \vdash_L e \Rightarrow v) \Longleftrightarrow (k :: \mathrm{nil} \vdash_H e \Rightarrow v)$.

Theorem 2. *For all $k \in Cont_L$, $e \in Exp_L$ and $v \in Val_L$, if $k \vdash_L e \Rightarrow v$, then $\forall\, ks \in Cont_L^*$, $v' \in Val_L$. $(\Box :: ks \vdash_H v \Rightarrow v') \Longrightarrow (k :: ks \vdash_H e \Rightarrow v')$.*

Proof. By rule induction on $k \vdash_L e \Rightarrow v$.

For $ks = \mathrm{nil}$, using rule $\mathsf{VAL0_H^{NIL}}$, we obtain the following corollary.

Corollary 2. *For all $k \in Cont_L$, $e \in Exp_L$ and $v \in Val_L$, if $k \vdash_L e \Rightarrow v$ then $k :: \mathrm{nil} \vdash_H e \Rightarrow v$.*

$$(\mathsf{VAL0}^{\mathsf{NIL}}_{\mathsf{H}})\frac{}{\Box :: \mathrm{nil} \vdash_H v \Rightarrow v} \qquad (\mathsf{VAL0}^{\mathsf{CONS}}_{\mathsf{H}})\frac{k :: ks \vdash_H v' \Rightarrow v}{\Box :: k :: ks \vdash_H v' \Rightarrow v}$$

$$(\mathsf{VAL1}_{\mathsf{H}})\frac{\Box :: ks \vdash_H k[v_1] \Rightarrow v}{k :: ks \vdash_H v_1 \Rightarrow v}\ (k \neq \Box) \quad (\mathsf{FN}_{\mathsf{H}})\frac{k[\Box\, e_1] :: ks \vdash_H e_0 \Rightarrow v}{k :: ks \vdash_H e_0\, e_1 \Rightarrow v}\ (e_0 \text{ not a value})$$

$$(\mathsf{ARG}_{\mathsf{H}})\frac{k[v_0\, \Box] :: ks \vdash_H e_1 \Rightarrow v}{k :: ks \vdash_H v_0\, e_1 \Rightarrow v}\ (e_1 \text{ not a value}) \quad (\mathsf{BETA}_{\mathsf{H}})\frac{k :: ks \vdash_H [v_1/x]e \Rightarrow v}{k :: ks \vdash_H (\lambda x.e)\, v_1 \Rightarrow v}$$

$$(\mathsf{shift}_{\mathsf{H}})\frac{\Box :: ks \vdash_H (\lambda c.e)\, \lambda w.\mathbf{pushcc}(w, k) \Rightarrow v}{k :: ks \vdash_H \mathbf{shift}\ c.e \Rightarrow v} \quad (\mathsf{reset}_{\mathsf{H}})\frac{k :: ks \vdash_H \mathbf{pushcc}(e, \Box) \Rightarrow v}{k :: ks \vdash_H \mathbf{reset}\ e \Rightarrow v}$$

$$(\mathsf{pushcc}_{\mathsf{H}})\frac{k :: k' :: ks \vdash_H e \Rightarrow v}{k' :: ks \vdash_H \mathbf{pushcc}(e, k) \Rightarrow v}$$

Table 3. An operational continuation semantics for **shift** and **reset**

For the inverse direction, we need to refer to the derivation more explicitly: we use $\preceq$ to denote the sub-derivation relation, and we write $D : J$ if D is a derivation ending with the judgment J.

Theorem 3. *For all* $ks \in Cont^*_L$, $k \in Cont_L$, $e \in Exp_L$, $v' \in Val_L$, *if* $D : k :: ks \vdash_H e \Rightarrow v'$, *then*

$$\exists\, v \in Val_L,\ D'.\ (k \vdash_L e \Rightarrow v) \wedge (D' \preceq D) \wedge D' : (\Box :: ks \vdash_H v \Rightarrow v').$$

Proof. By strong induction on the derivation D.

For $ks = \mathrm{nil}$ in Theorem 3, we notice that the only possible derivation for D' is one-step, using rule $\mathsf{VAL0}^{\mathsf{NIL}}_{\mathsf{H}}$, so the witness v is v', and the following corollary holds.

Corollary 3. *For all* $k \in Cont_L$, $e \in Exp_L$ *and* $v \in Val_L$, *if* $k :: \mathrm{nil} \vdash_H e \Rightarrow v$ *then* $k \vdash_L e \Rightarrow v$.

Together, Corollary 2 and Corollary 3 show that the branchless semantics H simulates the semantics L. H can be implemented by a definitional interpreter as before [5]. Our goal, however, is to implement the control operators natively as functions (using first-class continuations and cells, like Filinski [12]). The semantics of the built-in constructs cannot be altered in such a setting, thereby preventing us to implement the crucial rule $\mathsf{VAL0}^{\mathsf{CONS}}_{\mathsf{H}}$, which is enacted when the active continuation (of type $Cont_L$) is already identity. Such behavior should be put into the continuation: instead of initializing it with an identity continuation $\Box$, we should initialize it with an operation to resume the top continuation from the stack. The corresponding transformation of L is described in Section 2.7.

Now we can come back to the typing problem of **shift**. The requirement that all **shift**-expressions must be enclosed by a corresponding **reset** or **shift** can be easily manifested in the semantics H by adding a side condition to the rule

$$(\mathsf{VAL0_I})\frac{}{\Box :: ks \vdash_I v \Rightarrow v} \qquad (\mathsf{VAL1_I})\frac{\Box :: ks \vdash_I k[v_1] \Rightarrow v}{k :: ks \vdash_I v_1 \Rightarrow v}\ (k \neq \Box)$$

$$(\mathsf{FN_I})\frac{k[\Box\, e_1] :: ks \vdash_I e_0 \Rightarrow v}{k :: ks \vdash_I e_0\, e_1 \Rightarrow v}\ (e_0 \text{ not a value})$$

$$(\mathsf{ARG_I})\frac{k[v_0\, \Box] :: ks \vdash_I e_1 \Rightarrow v}{k :: ks \vdash_I v_0\, e_1 \Rightarrow v}\ (e_1 \text{ not a value}) \qquad (\mathsf{BETA_I})\frac{k :: ks \vdash_I [v_1/x]e \Rightarrow v}{k :: ks \vdash_I (\lambda x.e)\, v_1 \Rightarrow v}$$

$$(\mathsf{shift_I})\frac{id_L^{pop} :: ks \vdash_I (\lambda c.e)\, \lambda w.\mathbf{pushcc}(w, k) \Rightarrow v}{k :: ks \vdash_I \mathbf{shift}\ c.e \Rightarrow v}\ (ks \neq \text{nil})$$

$$(\mathsf{reset_I})\frac{k :: ks \vdash_I \mathbf{pushcc}(e, id_L^{pop}) \Rightarrow v}{k :: ks \vdash_I \mathbf{reset}\ e \Rightarrow v} \qquad (\mathsf{pushcc_I})\frac{k :: k' :: ks \vdash_I e \Rightarrow v}{k' :: ks \vdash_I \mathbf{pushcc}(e, k) \Rightarrow v}$$

$$(\mathsf{popcc_I})\frac{ks \vdash_I v \Rightarrow v'}{k' :: ks \vdash_I \mathbf{popcc}(v) \Rightarrow v'}\ (ks \neq \text{nil})$$

Table 4. An implementable semantics for **shift** and **reset**

$\mathsf{shift_H}$ that the stack should not be empty. (This makes the definition of **shift** partial; when the stack is empty, we obtain a run-time error.)

$$(\mathsf{shift_H})\frac{\Box :: ks \vdash_H (\lambda k'.e)\, \lambda w.\mathbf{pushcc}(w, k) \Rightarrow v}{k :: ks \vdash_H \mathbf{shift}\ k'.e \Rightarrow v}\ (ks \neq \text{nil})$$

2.7 An implementable continuation semantics for shift and reset

The implementable semantics I is also induced from the semantics L with $Cont_I = Cont_H = Cont_L^+$. We introduce a new operator **popcc**:

$$e ::= ... \mid \mathbf{popcc}(e)$$

Intuitively, this operator pops the continuation stack and sends its operand to the popped continuation. Now, the initial continuation can be defined as $id_L^{pop} \stackrel{\text{def}}{=} \mathbf{popcc}(\Box)$, which replaces $\Box$ in rules $\mathsf{shift_H}$ and $\mathsf{reset_H}$, thus eliminating the need for the rule $\mathsf{VAL0_H^{CONS}}$.

In Table 4, the I-rules are the same as the H-rules except for $\mathsf{VAL0_I}$ (replacing $\mathsf{VAL0_H^{NIL}}$ and $\mathsf{VAL0_H^{CONS}}$), $\mathsf{shift_I}$, $\mathsf{reset_I}$, and $\mathsf{popcc_I}$.

Semantics I simulates semantics L, as shown by the following theorem.

Theorem 4. *For all* $e \in Exp_L$ *and* $v \in Val_L$, $(\Box :: \text{nil} \vdash_I e \Rightarrow v) \Longleftrightarrow (\Box :: \text{nil} \vdash_H e \Rightarrow v)$.

Proof. By two straightforward inductions.

This new semantics I has two properties:

(1) It is branchless. More specifically, for any intermediate judgment $k \vdash_I e \Rightarrow v$ in a proof tree, the rest of the computation after e is evaluated is totally captured in the global continuation k. We can thus iterate the above process to add control operators at subsequent levels.

(2) It can be directly implemented in the starting semantics S using references and first-class continuations in the following way:[2] the head of the global continuation k is the current continuation, while the tail is stored in a reference cell. All the S-rules automatically 'extend' to the corresponding I-rules, without touching the reference cell; the new rules defining the control operators can then be directly implemented by encoding the four constants **shift**, **reset**, **pushcc** and **popcc** as functions, using **callcc** to capture the current continuation and **throw** to restore it.

2.8 An inductive construction of the CPS hierarchy

The semantic transformation (from Section 2.5 to Section 2.7) can be generalized and iterated: at each step, we transform an input semantics S_i into an output semantics S_{i+1} that preserves certain *inductive conditions* (such as "branchlessness"). The operational continuation semantics displayed in Table 1 satisfies these inductive conditions, and we used it as the starting semantics S_1.

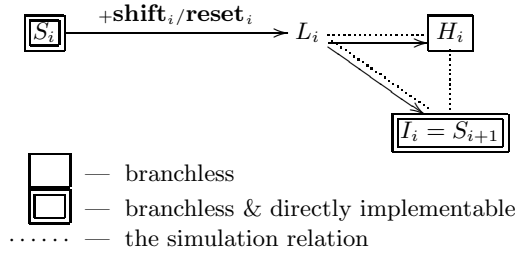


Fig. 1. A single transformation (from level i to level $i + 1$)

Figure 1 summarizes the development of a single transformation: we start with the branchless semantics S_i of level i. Adding **shift**$_i$ and **reset**$_i$ with related inference rules yields the semantics L_i which is no longer branchless (see Section 2.5). Then we replace the global continuation of this level by a stack of such continuations, which forms the global continuation of the next level, and we obtain a semantics H_i where we restore the branchlessness property (see Section 2.6). Since H_i is not directly implementable in S_i, we apply another transformation to L_i to obtain a semantics I_i, which is both branchless and directly implementable (see Section 2.7). Semantics I_i simulates H_i, which in turn simulates L_i. With its newly introduced control operators **shift**$_i$ and **reset**$_i$, I_i satisfies the inductive conditions. Therefore, we can use it as the semantics S_{i+1} for the next level of the hierarchy.

[2] We did not put references and **callcc** in the starting semantics and we only use them for the implementation. In fact, making references available to the user causes no problem, whereas **callcc** interferes with **shift** and **reset**.

For space reasons, the rest of this section is omitted. But it is available in the extended version of this article [7].

3 An Implementation of the CPS Hierarchy

We implement the CPS hierarchy by transcribing the transformation of Section 2.8 in Standard ML of New Jersey [1], using the structure `SMLofNJ.Cont` which provides `callcc` and `throw`. The implementation uses a signature `SHIFT_ RESET` to specify the operations provided by a semantics S for the user and for the construction of the next control level, a structure `innermost_level` to model the first level in the hierarchy (which thus provides no control operators), and a functor `sr_outer` to construct next-level control operators, parameterized by the answer type `ans` and by the control level `inner` immediately preceding it. Essentially, the signature `SHIFT_RESET` corresponds to the inductive conditions for the semantics S, and the functor `sr_outer` corresponds to the transformation from a semantics $S = S_i$ to the next-level semantics $I = S_{i+1}$.

The implementations of control operators are thus hidden inside the module system, and they are accessed via the name of the structure that corresponds to their level. Having devised an ordering of the control effects, a user then implements it through the order of functor applications. Hierarchical occurrences of **shift** and **reset** are thus no longer referred to by their relative index, which had been criticized in the literature [16,27].

We implemented the functor `sr_outer` by transcribing line-by-line the added semantic rules in semantics I (four new functions, one per operator) and the definition of the constant id_L^{pop}.[3] We also use two auxiliary functions: a function `replace_gcont`, used implicitly in the semantics, captures and replaces the current global continuation, and a function `cont2gcont`, required by the inductive conditions for semantics S, converts a first-class continuation to a global continuation. The code is thus very concise: the pretty-printed program defining `innermost_level` and `sr_outer` takes about 40 lines of ML code (Figure 3).

We also provide a functor for the usual first level of control operators ($\mathbf{shift}_1$ and $\mathbf{reset}_1$):

```
functor initial_control_level (type ans) : SHIFT_RESET
= sr_outer (type ans = ans  structure inner = innermost_level)
```

Specializing this functor for the first level of the CPS hierarchy yields a result similar to Filinski's implementation of **shift** and **reset** [12]. The main difference is that here we use an explicit stack of continuations whereas Filinski uses an implicit one through functional abstraction. (An analogy: one can represent environments in an interpreter as a list or as a function.)

[3] We use the function `SMLofNJ.Cont.isolate` to coerce a non-returning function to a continuation. This function can be defined as follows.

```
fun isolate f = callcc (fn x => f (callcc (fn y => throw x y)))
```

```
signature SHIFT_RESET                                              (* control level i *)
= sig
    type answer                                                    (* answer type of level i *)
    val reset : (unit -> answer) -> answer
    val shift : (('a -> answer) -> answer) -> 'a
    type 'a gcont                                                  (* Cont_S^i (= Cont_L^i) *)
    val replace_gcont : 'a gcont -> ('b gcont -> 'a) -> 'b
                                (* captures current global continuation (of type 'b gcont), *)
                              (* and replaces it with the first argument (of type 'a gcont) *)
    val cont2gcont : 'a cont -> 'a gcont                           (* Cont_S^0 -> Cont_S^k *)
  end

structure innermost_level :> SHIFT_RESET                                          (* level 0 *)
= struct                                  (* here, global continuation = ML continuation *)
    exception InnermostLevelNoControl
    type answer = unit
    type 'a gcont = 'a cont                          (* uses ML continuation for Cont_S^0 *)

    fun replace_gcont new_c e_thunk
        = callcc (fn old_c => throw new_c (e_thunk old_c))
    fun cont2gcont c = c
    fun reset _ = raise InnermostLevelNoControl
    fun shift _ = raise InnermostLevelNoControl
  end

functor sr_outer (type ans structure inner: SHIFT_RESET) :> SHIFT_RESET
where type answer = ans                                  (* from S = S_i to I = S_{i+1} *)
= struct
    exception MissingReset
    exception Fatal
    type answer = ans
    type 'a gcont                                                  (* Cont_I = Cont_S^+ *)
         = (answer inner.gcont) list * 'a inner.gcont
    val stack = ref [] : (answer inner.gcont) list ref                              (* ks *)

    fun replace_gcont (new_ks, new_k) e_thunk
                            (* captures and replaces the global continuation, recursively *)
        = inner.replace_gcont new_k
            (fn cur_k => let val cur_gcont = (!stack, cur_k)
                         in stack := new_ks; (e_thunk cur_gcont) end)
    fun cont2gcont action
        = ([], inner.cont2gcont action)
    fun popcc v                                                    (* rule popcc_I *)
        = case !stack of                                  (* side condition (ks ≠ nil) *)
            [] => raise Fatal
          | k'::ks => (stack := ks; inner.replace_gcont k' (fn _ => v))
    val id_popcc = inner.cont2gcont (isolate popcc)                        (* id_L^pop *)
    fun pushcc k e_thunk                                           (* rule pushcc_I *)
        = inner.replace_gcont k
            (fn k' => (stack := k' :: !stack; e_thunk ()))
    fun reset e_thunk                                              (* rule reset_I *)
        = pushcc id_popcc e_thunk
    fun shift k_abstraction                                        (* rule shift_I *)
        = case !stack of                                  (* side condition (ks ≠ nil) *)
            [] => raise MissingReset
          | _ => inner.replace_gcont id_popcc
                    (fn (k : 'a inner.gcont)
                     => k_abstraction (fn w => pushcc k (fn () => w)))
end
```

Fig. 2. A native implementation of the CPS hierarchy in Standard ML of New Jersey

4 Application: layering monadic effects

As a significant application of composable continuations, Filinski's work on adding user-defined monadic effects to ML-like languages by *monadic reflection* shows that composing continuations is a universal effect, which can be used to simulate all effects expressible using a monad [12,13]. The original work only allowed one monadic effect, but recently, Filinski has extended the technique to allow layering effects by relating a heterogeneous tower of monads to a tower of continuation monads, and then implementing them using a collection of cells to hold the meta-continuations [15].

Independently, we directly adapted Filinski's original one-level implementation with minimal changes to parameterize the functor that generates a *monad representation* by the monad representation layered beneath it, which also gives an inductive implementation of a monadic hierarchy. We essentially put in each structure of a monad representation the corresponding level in the control hierarchy; the functor that generates an outer monad representation is passed the control level of the monad representation at the inner layer, and applies functor `sr_outer` to construct its own control level.

The benefits of this representation of layered monads is the same as in Filinski's work [15]: it is a direct implementation, i.e., no level of interpretation and no level of translation hinder it [26,36].

More detail and several illustrative examples are available in the extended version of this article [7].

5 Related Work

5.1 Felleisen's seminal work

As already mentioned in Section 1.2, the notion of control delimiters in direct style is due to Felleisen [8]. As already pointed out by Danvy and Filinski [5], control delimiters are significant because they fit in each level of the CPS hierarchy very naturally: they correspond to resetting the current continuation to the identity function; and indeed the control delimiter **reset** is equivalent to Felleisen's. As for abstracting control, programming practice suggested the control operator **shift** which is equivalent to one of the variants of Felleisen's $\mathcal{F}$-operator.

Felleisen's work relies on a notion of control stack, and has inspired a number of similar control operators. Danvy and Filinski's work relies on CPS, and has inspired a number of applications, for two compound reasons we believe:

Expressiveness: Programming intuitions run strong in the world of control stacks. But lacking guidelines, how does one know, e.g., whether one has landed on [the continuation equivalent of] Algol 60's control stack or on Lisp's control stack—i.e., on the control equivalent of lexical scope or on dynamic scope (whichever may be best)? And how does one use the result?

Conversely, the world of CPS is a structured one, which offers guidelines and holds much untapped expressive power. For example [12,13], Filinski has

shown that the expressive power of the CPS hierarchy is equivalent to the one of computational monads. In fact, our new examples could equally well be expressed using a tower of monads.

More specifically, operational descriptions of control hierarchies offer the possibilities to shadow control delimiters, to capture them or not when abstracting control, to restore them or not when reinstating abstracted control, and to dynamically search through them at run time. CPS shields us against the most extravagant of these mind-boggling possibilities, since by definition, programs with **shift** and **reset** denote CPS programs. These CPS programs may have many layers of continuations, but they are (1) purely functional and (2) statically typed.

Efficient implementation: A stack-based implementation of control tends to exert a cost which is linear in the use of each captured continuation. Besides, and this is a well-known thesis in the continuation community [3], it faces a real problem of duplicated continuations.

Therefore alternative implementations have been sought. For example, Filinski already showed that $\mathbf{shift}_1$ and $\mathbf{reset}_1$ can be implemented concisely in terms of **callcc**, which itself can be implemented efficiently [3,12,20]. Through an alternative (but equivalent) formalism, our work essentially generalizes this concise implementation to the whole CPS hierarchy, with no new cost and an equivalent use.

5.2 Filinski's work

As a significant application of the CPS hierarchy, Filinski's work on adding user-defined monadic effects to ML-like languages by *monadic reflection* shows that composing continuations is a universal effect, which can be used to simulate all effects expressible using a monad [12,13,15].

5.3 Gunter, Rémy, and Riecke's work

Gunter, Rémy, and Riecke present a new set of control operators generalizing exceptions and continuations, and its associated operational semantics and type system [16]. The strength of these operators lies in their static type system—in comparison, and even though we do not doubt that there is one for the CPS hierarchy (cf. Murthy's work [28]), we do not present one here explicitly; instead, we rely on ML's type system in our implementation.

Independently of their type system, Gunter, Rémy, and Riecke's operators are not cast in stone. In their own words, "We do not feel, though, that there is a clear answer to the question of which operational rule is right; suffice it to say that we have picked one, and that the other rules lead to strong type soundness as well." Similarly, we do not contend that **shift** and **reset** are the ultimate control operators—Filinski's operators **kreflect** and **kreify**, for example, could well be preferred [12,13]. But we do believe that the key to their simplicity and expressiveness is the CPS hierarchy.

Gunter, Rémy, and Riecke's operators are also implemented with **callcc**.

5.4 Operational semantics

Operational semantics, especially small-step reduction semantics, is often used to specify control operators formally. Several researchers have investigated the type soundness of languages with control operators via syntactic approaches based on operational semantics, such as Wright and Felleisen, and Harper, Duba, and MacQueen for first-class continuations [17,39], Gunter, Rémy, and Riecke for generalizing exceptions and continuations [16], and Murthy for the CPS hierarchy [28]. Here, we use operational semantics to derive our implementation and to prove its correctness. Also, matching the CPS hierarchy, we present a family of continuation semantics instead of one monolithic semantics. This family can be natively programmed in ML without resorting to an informal notion of control stack.

5.5 Continuations

After 25 years of existence [32], continuations still remain a challenging topic, to the point that ad-hoc frameworks are routinely preferred. For example, we find it significant that alternative and independent solutions were sought to compile goal-directed evaluation [30] and to abstract delimited control [8,16], even though two levels of continuations provide a simple, natural, and directly implementable solution to both problems. This indicates that continuations require more basic research. We have tried to contribute to this research by characterizing a specific notion of operational continuation semantics and by formalizing its connection to the traditional CPS transformation.

6 Conclusion

The CPS transformation is ubiquitous in many areas of computer science, including logic, constructive mathematics, programming languages, and programming. Iterating it yields a concise and expressive framework for delimiting and abstracting control—the CPS hierarchy—which appears substantial and fruitful but has been explored very little so far. In this article, we have contributed to exploring it by (1) characterizing an operational analogue of continuation semantics; (2) developing an analogue of the CPS transformation for such an operational continuation semantics; (3) making it account for the family of control operators **shift** and **reset**; (4) providing a native implementation of the CPS hierarchy in the statically typed language Standard ML; and (5) illustrating the implementation both with classical and with new applications, and in particular with a direct implementation of layered monads.

Acknowledgments

Part of this work was carried out while the second author was visiting the BRICS PhD school at the University of Aarhus in the fall of 1997.

Thanks are due to Andrzej Filinski, Julia Lawall, and the anonymous referees for comments on an earlier version of this article.

References

1. Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Jan Maluszyński and Martin Wirsing, editors, *Third International Symposium on Programming Language Implementation and Logic Programming*, number 528 in Lecture Notes in Computer Science, pages 1–13, Passau, Germany, August 1991. Springer-Verlag.
2. Robert (Corky) Cartwright, editor. *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, Snowbird, Utah, July 1988. ACM Press.
3. William Clinger, Anne H. Hartheimer, and Eric M. Ost. Implementation strategies for first-class continuations. *Higher-Order and Symbolic Computation*, 12(1), 1999.
4. Olivier Danvy. Type-directed partial evaluation. In Guy L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 242–257, St. Petersburg Beach, Florida, January 1996. ACM Press.
5. Olivier Danvy and Andrzej Filinski. Abstracting control. In Wand [38], pages 151–160.
6. Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992.
7. Olivier Danvy and Zhe Yang. An operational investigation of the CPS hierarchy (extended version). Technical Report BRICS RS-98-35, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 1998.
8. Matthias Felleisen. The theory and practice of first-class prompts. In Ferrante and Mager [11], pages 180–190.
9. Matthias Felleisen, Daniel P. Friedman, Bruce Duba, and John Merrill. Beyond continuations. Technical Report 216, Computer Science Department, Indiana University, Bloomington, Indiana, February 1987.
10. Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. Abstract continuations: A mathematical semantics for handling full functional jumps. In Cartwright [2], pages 52–62.
11. Jeanne Ferrante and Peter Mager, editors. *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1988. ACM Press.
12. Andrzej Filinski. Representing monads. In Hans-J. Boehm, editor, *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 446–457, Portland, Oregon, January 1994. ACM Press.
13. Andrzej Filinski. *Controlling Effects*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1996. Technical Report CMU-CS-96-119.
14. Andrzej Filinski. From normalization-by-evaluation to type-directed partial evaluation. In Olivier Danvy and Peter Dybjer, editors, *Preliminary Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation, NBE '98,* (Chalmers, Sweden, May 8–9, 1998), number NS-98-1 in BRICS Note Series, Department of Computer Science, University of Aarhus, May 1998.
15. Andrzej Filinski. Representing layered monads. In Alex Aiken, editor, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Programming Languages*, San Antonio, Texas, January 1999. ACM Press. To appear.
16. Carl Gunter, Didier Rémy, and Jon G. Riecke. A generalization of exceptions and control in ML-like languages. In Simon Peyton Jones, editor, *Proceedings of the*

Seventh ACM Conference on Functional Programming and Computer Architecture, pages 12–23, La Jolla, California, June 1995. ACM Press.
17. Robert Harper, Bruce F. Duba, and David MacQueen. Typing first-class continuations in ML. *Journal of Functional Programming*, 3(4):465–484, October 1993.
18. Robert Hieb and R. Kent Dybvig. Continuations and concurrency. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, SIGPLAN Notices, Vol. 25, No. 3, pages 128–136, Seattle, Washington, March 1990. ACM Press.
19. Robert Hieb, R. Kent Dybvig, and Claude W. Anderson, III. Subcontinuations. *Lisp and Symbolic Computation*, 5(4):295–326, December 1993.
20. Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In Bernard Lang, editor, *Proceedings of the ACM SIGPLAN'90 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 25, No 6, pages 66–77, White Plains, New York, June 1990. ACM Press.
21. Gregory F. Johnson. GL – a denotational testbed with continuations and partial continuations as first-class objects. In Mark Scott Johnson, editor, *Proceedings of the ACM SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques*, SIGPLAN Notices, Vol. 22, No 7, pages 154–176, Saint-Paul, Minnesota, June 1987. ACM Press.
22. Gregory F. Johnson and Dominic Duggan. Stores and partial continuations as first-class objects in a language and its environment. In Ferrante and Mager [11], pages 158–168.
23. Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998.
24. Julia L. Lawall. *Continuation Introduction and Elimination in Higher-Order Programming Languages*. PhD thesis, Computer Science Department, Indiana University, Bloomington, Indiana, July 1994.
25. Julia L. Lawall and Olivier Danvy. Continuation-based partial evaluation. In Carolyn L. Talcott, editor, *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. VII, No. 3, Orlando, Florida, June 1994. ACM Press.
26. Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In Peter Lee, editor, *Proceedings of the Twenty-Second Annual ACM Symposium on Principles of Programming Languages*, pages 333–343, San Francisco, California, January 1995. ACM Press.
27. Luc Moreau and Christian Queinnec. Partial continuations as the difference of continuations, a duumvirate of control operators. In Manuel Hermenegildo and Jaan Penjam, editors, *Sixth International Symposium on Programming Language Implementation and Logic Programming*, number 844 in Lecture Notes in Computer Science, pages 182–197, Madrid, Spain, September 1994. Springer-Verlag.
28. Chethan R. Murthy. Control operators, hierarchies, and pseudo-classical type systems: A-translation at work. In Olivier Danvy and Carolyn L. Talcott, editors, *Proceedings of the ACM SIGPLAN Workshop on Continuations*, Technical report STAN-CS-92-1426, Stanford University, pages 49–72, San Francisco, California, June 1992.
29. Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

30. Todd A. Proebsting. Simple translation of goal-directed evaluation. In Ron K. Cytron, editor, *Proceedings of the ACM SIGPLAN'97 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 32, No 5, pages 1–6, Las Vegas, Nevada, June 1997. ACM Press.
31. Christian Queinnec and Bernard Serpette. A dynamic extent control operator for partial continuations. In Robert (Corky) Cartwright, editor, *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 174–184, Orlando, Florida, January 1991. ACM Press.
32. John C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3/4):233–247, December 1993.
33. Dorai Sitaram. Handling control. In David W. Wall, editor, *Proceedings of the ACM SIGPLAN'93 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 28, No 6, pages 147–155, Albuquerque, New Mexico, June 1993. ACM Press.
34. Dorai Sitaram and Matthias Felleisen. Control delimiters and their hierarchies. *Lisp and Symbolic Computation*, 3(1):67–99, January 1990.
35. Dorai Sitaram and Matthias Felleisen. Reasoning with continuations II: Full abstraction for models of control. In Wand [38], pages 161–175.
36. Philip Wadler. The essence of functional programming (tutorial). In Andrew W. Appel, editor, *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, January 1992. ACM Press.
37. Philip Wadler. Monads and composable continuations. *LISP and Symbolic Computation*, 7(1):39–55, January 1994.
38. Mitchell Wand, editor. *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, June 1990. ACM Press.
39. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.

Higher-Order Code Splicing

Peter Thiemann*

Institut fü¸ Informatik
Universität Freiburg, Germany
thiemann@informatik.uni-freiburg.de

Abstract. Run-time code generation (RTCG) and just-in-time compilation (JIT) are features of modern programming systems to strike the balance between generality and efficiency. Since RTCG and JIT techniques are not portable and notoriously hard to implement, we propose code splicing as an alternative for dynamically-typed higher-order programming languages. Code splicing combines precompiled pieces of code using higher-order functions. While this approach cannot achieve the performance of compiled code, it can support some intriguing features:

- very fast "compilation" times;
- satisfactory run times, compared with interpretation;
- simple interfacing with compiled code;
- portability.

Starting from implementation models for functional languages we develop and evaluate several approaches to code splicing. This leads to some new insights into compilation techniques for functional programming languages, among them a compositional compilation schema to SKI-combinators. The progression of different techniques sheds some light on their relationship, specifically between combinator-based implementations and closure-based implementations.
All techniques have been implemented and evaluated in Scheme.

1 Introduction

Run-time code generation and just-in-time compilation generate code at run time and execute it subsequently. To amortize code generation time, RTCG and JIT only perform simple optimizations. They can still generate competitive code by exploiting invariants that are only available at run time. On the flip-side these techniques are inherently non-portable and hard to implement because many technical details and pitfalls must be catered for (flushing instruction caches, memory management, access permissions, and so on).

However, there is some indication that RTCG is not always worth the effort. Indeed, Lee [17] stated that restructuring the code to observe a staging discipline already achieves significant speedups. So why go into the complications of RTCG or JIT if some of the speedup is already available just by staging?

* This work has been done while at the School of Computer Science and Information Technology, University of Nottingham, UK. The author acknowledges support by EPSRC grant GR/M22840 "Semantics of Specialization".

S.D. Swierstra (Ed.): ESOP/ETAPS'99, LNCS 1576, pp. 243–257, 1999.
© Springer-Verlag Berlin Heidelberg 1999

This is exactly the motivation for code splicing. Code splicing is a range of techniques for writing staged interpreters. Such an interpreter keeps compile-time computations separate from run-time computations. Applying it to a source program ideally performs all compile-time computations and returns the compiled (or code-spliced) program. It is not necessary to actually build a compiler. This style of "compilation" is attractive for a range of applications:

- The execution of applets and other mobile code: receive high-level code from a network to perform integrity checks, but execute it efficiently.
  Many applets are throw-away code which is only executed a very limited number of times. Even JIT compilation may be too expensive [2].
- The implementation of flexible domain-specific languages.
  The emphasis is on quickly designing, implementing, and (possibly) modifying the language. Writing a full-blown compiler would be too expensive because the user community for such a language is often small and efficiency is not of tantamount importance.
- Efficient metaprogramming.
  Metaprogramming systems [20] allow the generation of high-level program code and its subsequent execution in the same running program. Ideally, there should be no penalty for using generated code, but full compilation would be too slow. The approaches to combining partial evaluation and compilation also fall into this category [19, 5].
- Overcoming restrictions of compilers.
  Some compilers have arbitrary restrictions on the size of code, the number of variables, and so on. Code splicing provides a way to compile and execute arbitrary programs, overcoming implementation restrictions.

All these applications share the following requirements:

1. *Instantaneous compilation.* The time for code splicing should be comparable to the time necessary to construct the corresponding source text.
2. *Satisfactory speed.* Code-spliced programs should be significantly faster than interpreted ones.
3. *Easy interfacing.* It should be possible to freely mix code-spliced and ordinarily compiled code.
4. *Portability.*

The present paper investigates several code splicing techniques through implementations in the higher-order functional programming language Scheme [15]. All techniques are portable and most of them meet all of the requirements. They exploit many features of Scheme, e.g., dynamic typing, side effects, and the `eval` function.

The techniques are inspired by implementation techniques for functional programming languages (SKI-combinators, director strings, categorical combinators, closure conversion, and shallow binding). Most of them can be made to exhibit good staging properties. However, there are some surprises. For example, it turns out that the standard compilation method to SKI-combinators [21]

```
ASyntax        type of generated code (active syntax)
make-var     : Var → ASyntax
make-lam     : Var × ASyntax → ASyntax
make-app     : ASyntax × ASyntax → ASyntax
compile-term : ASyntax → Val
```

Fig. 1. Signature of code generation

is unsuitable for our purposes because it is not compositional. Therefore, we devise a compositional compilation scheme to SKI-combinators and prove some correctness properties about it. Deforestation [22] makes it reasonably efficient.

Outline: Section 2 introduces our experimental setup. Section 3 explains the different approaches to compilation that we consider. Section 4 presents comparative run times. Section 5 discusses related work and Section 6 concludes.

Throughout the paper, we assume knowledge of the Scheme language [15].

2 The experimental setup

Type-directed partial evaluation (TDPE) [7] is our dynamic supply for program text. When applied to a value of type τ and a representation of the type τ, it constructs a pure lambda term of that type. To "compile" the constructed terms using code splicing, it is sufficient to provide staged interpreters for the lambda calculus. We distribute the implementation of such an interpreter over the syntax constructors for the lambda calculus to avoid the cost of the syntax dispatch. That is, the `make-var` function interprets a variable expression, the `make-lam` function interprets a lambda abstraction, and `make-app` implements function application, taking the interpreted subexpressions as parameters. Fig. 1 defines the types of these functions. Each interpreter provides its own definition of the type `ASyntax` (for active syntax) and the functions listed, the constructors for this type. The implementation includes multi-argument versions `make-lam*` and `make-app*` of lambda abstraction and application, as well as `make-let`.

3 Approaches to code splicing

The following subsections introduce different ways to achieve code splicing. Each choice distributes compile-time work between the active syntax constructors and `compile-term` in a different way. In one extreme (see Sec. 3.1), the active syntax constructors construct Scheme expressions and `compile-term` performs full compilation. In the other extreme [19], the active syntax constructors emit and combine byte code and `compile-term` is the identity function.

```
(define (make-var x)     x)
(define (make-lam x e)   `(LAMBDA (,x) ,e))
(define (make-app f a)   `(,f ,a))
(define (compile-term e) (eval e (interaction-environment)))
```

Fig. 2. Active syntax constructors for Scheme

3.1 Using `eval`

The Scheme standard [15] includes a function `eval` that maps a Scheme expression to its value. Hence, the syntax constructors can construct Scheme expressions (using `quasiquote` "`" and `unquote` ",") and `compile-term` can be `eval` (see Fig. 2).

Unfortunately, this implementation does not fulfill all of our requirements: compilation speed is heavily implementation and system dependent. Furthermore, the compiled code is likely to uncover implementation restrictions of the underlying language implementation.

3.2 SKI combinators

This approach takes advantage of a precompiled library of implementations of the combinators S, K, and I. After compiling the source program to a combinator term, an interpreter just sticks the precompiled combinators together as prescribed by this term.

Unfortunately, the naive compilation generates abysmal code and requires multiple passes. An optimized compilation [21] generates better and smaller code, but it still requires multiple passes. However, we can do better.

Compositional compilation to combinators Some research has gone into optimized combinator systems that keep the resulting terms small [21]. If we adopt the additional combinators B and S' defined by

B = `(lambda (x) (lambda (y) (lambda (z) (x (y z)))))`
S' = `(lambda (k) (lambda (x) (lambda (y) (lambda (z) ((k (x z)) (y z))))))`

then there is a compositional specification of compilation. Let us consider the three constructs in turn for `ASyntax` = `CEnv` $\rightarrow$ `SKI` where the compile-time environment `CEnv` = `Var`* is the list of the bound variables in the reverse order in which they were bound and `SKI` is the set of combinator terms.

`make-var`. An access to the ith variable in an environment of size n compiles into a projection function that returns the ith argument out of n (where $0 \leq i < n$ and $n > 0$): $\lambda x_0 \ldots \lambda x_{n-1}.x_i$. This function can be expressed as $K^{(i)}((BK)^{(n-i-1)}(I))$ where $X^{(0)} = I$ and $X^{(i+1)} = BX^{(i)}X$ is the iterated composition of X (proof by case analysis and induction). Using the equivalent definition $X^{(0)}Y = Y$ and $X^{(i+1)}Y = X^{(i)}(XY)$, we get for $i = 1$ and $n = 3$ the combinator term $K(BKI)$.

`make-lam`. Compilation just needs to update the compile-time environment with the new variable. The translation of the body provides the additional abstraction.

`make-app`. If n is the size of the environment, the combinator

$$S_n f a = \lambda x_0 \ldots \lambda x_{n-1}.(f x_0 \ldots x_{n-1})(a x_0 \ldots x_{n-1})$$

distributes n values to the compiled subterms f and a of the application. Rewriting this combinator using only S, K, and I leads to an explosion of the size of the term. However, the S' combinator was conceived to solve this problem. It is easy to show that, for all $n \geq 0$, $S_n = (S'^{(n)}(I))$ (proof by induction).

Assessment The resulting combinator terms can be quadratic in the size of the source term (because the size n of the environment is only bounded by the size of the source term), hence the compile time is also at least quadratic.

In addition, the granularity of the compiled code is too fine, leading to disappointing performance.

Finally, dealing with multi-argument functions is awkward. The standard solution [10, sec. 12.2.3] is to introduce an untupling combinator

$$U = \texttt{(lambda (f) (lambda (x . xs) (apply (f x) xs)))}$$

and implement the translation of multi-argument abstractions accordingly.

Director strings The key idea of this approach is the following invariant:

pass only the values of the free variables to the compiled expression.

For this optimization, we need further combinators B', C', and C'' defined by

$B' =$ `(lambda (k) (lambda (x) (lambda (y) (lambda (z) ((k x) (y z))))))`
$C' =$ `(lambda (k) (lambda (x) (lambda (y) (lambda (z) ((k (x z)) y)))))`
$C'' =$ `(lambda (k) (lambda (x) (lambda (y) (lambda (z) ((k (x y)) z)))))`

During compilation, the compiler trims the environment according to the free variables while keeping their order. This idea corresponds exactly to director strings [16], where S', B', and C' correspond to Λ, \, and /. The `make-`***xxx*** functions compute the free variables on the fly at compile time.

`(make-var x)`. By the invariant, only the value of `x` is passed to this compiled expressions. Hence the translation is I.

`e` $\equiv$ `(make-lam x b)`. If `x` occurs free in `b` then the free variables of `b` are exactly `x` and the free variables of `e`: nothing needs to be done.

If `x` does not occur free in `b` then the free variables of `b` are identical to the free variables of `e`. In this case an additional abstraction must be conjured up. If n is the number of free variables and A is the compiled body then $C''^{(n)}KA$ does the job, since, for all $n \geq 0$,

$$\lambda x_0 \ldots \lambda x_n.A x_0 \ldots x_{n-1} = C''^{(n)}KA.$$

```
(define (make-var x)
  (lambda (env)
    (let* ((n (length env))
           (i (index x env)))
      (projection i n))))

(define (make-lam x e)
  (lambda (env)
    (e (cons x env))))

(define (make-app e1 e2)
  (lambda (env)
    (let* ((c1 (e1 env))
           (c2 (e2 env))
           (n (length env)))
      ((application n) c1 c2))))

(define (compile-term e1)
  (e1 '()))
```

Fig. 3. Active syntax constructors: customized combinators

`e` $\equiv$ `(make-app f a)`. This case involves bookkeeping of the free variables of `f`, `a`, and `e`. Let $x_1, \ldots, x_n$ be the free variables of `e` in the order of their bindings and compile to $X_1(X_2 \ldots (X_n I))FA$, where F and A are the compiled terms for f and a and, for $1 \leq i \leq n$,

- $X_i \equiv S'$ if $x_i \in FV(\mathtt{f}) \cap FV(\mathtt{a})$;
- $X_i \equiv C'$ if $x_i \in FV(\mathtt{f}) \setminus FV(\mathtt{a})$;
- $X_i \equiv B'$ if $x_i \in FV(\mathtt{a}) \setminus FV(\mathtt{f})$.

Deforestation Our implementation uses deforestation twice to eliminate intermediate results. First, the active syntax construction is re-interpreted as compilation to a combinator expression. Second, the syntax constructors of the resulting combinator expression (combinator constants and application) are re-interpreted as the compiled values of the combinators and as function application.

3.3 Customized combinators

A possible improvement to using SKI combinators lies in providing precompiled projection functions `(projection i n)` (with `n` > 0 and $0 \leq$ `i` $<$ `n`) and a generalized S combinator `(application n)` that distributes `n` ≥ 0 arguments.

```
(define (projection i n)
  (lambda (x0) ... (lambda (xn-1) xi)))
(define (application n)
  (lambda (f) (lambda (a)
    (lambda (x0) ... (lambda (xn-1)
      (((f x0) ... xn-1)
       ((a x0) ... xn-1)))))))
```

With this approach, the active syntax constructors perform the entire compilation, as shown in Fig. 3. The type of the active syntax is still `CEnv` $\rightarrow$ `Val` where `CEnv` is as before. The translation of a variable selects a projection using the size of `CEnv` and the index of the variable in it. The translation of a lambda abstraction only pushes the new variable onto the environment, it does not generate code. `Make-app` compiles its subexpressions first. Then it splices the two pieces of

```
(define (make-var x)
  (lambda (env)
    (let ((j (index x env)))
      (lambda vs
        (list-ref vs j)))))

(define (make-lam x e)
  (lambda (env)
    (let ((c (e (cons x env))))
      (lambda vs
        (lambda (y)
          (apply c (cons y vs)))))))

(define (make-app e1 e2)
  (lambda (env)
    (let* ((c1 (e1 env))
           (c2 (e2 env)))
      (lambda xs
        ((apply c1 xs)
         (apply c2 xs))))))

(define (compile-term e)
  ((e '())))
```

Fig. 4. Active syntax constructors: multi-argument combinators

code together to an application using `application`. The function `compile-term` initiates compilation by applying the value to the empty environment `'()`.

This compilation scheme exhibits linear time behavior in practice. Asymptotically it is quadratic due to the linear scans through the environment, whose length is also bounded by the size of the expression.

There are at least three ways to implement this approach in Scheme.

1. Generate the text of the `projection` and `application` functions as required, compile them using `eval`, and cache the results.
2. Use generic versions of `projection` and `application` in Scheme.
3. A mixed approach provides the precompiled versions up to some fixed n_0, falling back to a generic implementation for $n > n_0$.

A drawback is the complicated treatment of multi-argument functions which must be resolved in the same way as explained in Sec. 3.2.

3.4 Multi-argument combinators

The combinator approaches described in the preceding sections represent the run-time environment implicitly using abstractions "built into" the combinators S, K, and I, and generalizations thereof. This representation corresponds roughly to a list of the values of the variables, i.e., a nested representation of closures. It is inefficient because the free variables are passed one by one at each application and variable access.

The next logical step consists of flattening these structures and passing them around using multi-argument functions. With this approach, one application passes the entire run-time environment. Consequently, the type of the active syntax is $\prod \mathtt{env} : \mathtt{Var}^*.\mathtt{Val}^{|\mathtt{env}|} \to \mathtt{Val}$ where $|\mathtt{env}|$ is the size of the environment. The staging is important: compilation executes the $\prod \mathtt{env} : \mathtt{Var}^* \ldots$ part, whereas the $\mathtt{Val}^{|\mathtt{env}|} \to \mathtt{Val}$ part is left till run-time.

Figure 4 shows the interpretation of the constructors. The variable case computes the index `j` of the variable in the run-time environment and returns the

generic projection function `(lambda vs (list-ref vs j))` which maps a tuple of size at least `j+1` to the value of its `j`th component. `Make-lam` transfers the argument value `y` into the run-time environment. Effectively, it curries the compiled body `c` of the abstraction. `Make-app` is a generalized S combinator.

It is intriguing to see that we can directly map to compiled code so that not much work is left for `compile-term`: it applies active syntax to the empty environment and runs the resulting thunk (a parameterless function).

The code uses generic implementations of the `projection` and `application` functions (which is viable due to the use of the flat representation of the run-time environment). It is again possible to generate customized versions for each particular arity and cache their compiled code as explained above. In this case, we also need currying functions for different sizes of the run-time environment. To our surprise, we found that the cost of caching was higher than the cost of using the generic implementation. We suspect that checking the number of arguments for functions of fixed arity is the culprit: the generic implementation uses variadic functions that do not check the number of their arguments.

3.5 Explicit linked-list run-time environment

Another variation is inspired by the implementation technique of the categorical abstract machine [6]. Standard expository texts [11] employ the same technique.

The type of the active syntax is `CEnv` $\rightarrow$ `REnv` $\rightarrow$ `Val`. The representation of the run-time environment `REnv` is a linked list of vectors. Each vector holds the values of the variables abstracted by one surrounding lambda. The compile-time environment maps a variable to a depth and an offset, the depth determines the index in the outer linked list and the offset determines the position of the variable's value in the vector. Hence, `make-var` relies on cached projection functions, one for each pair of depth and offset.

`Make-lam` is implemented in terms of its multi-argument cousin `make-lam*`:

```
(define (make-lam* xs e)
  (lambda (env)
    (let ((comp (e (cons xs env))))
      (lambda (rt-env)
        (lambda ys
          (comp (cons (list->vector ys) rt-env)))))))
```

`Make-lam*` is staged: It performs all computations that only depend on the compile-time environment `env` before it abstracts the run-time environment `rt-env`. The run-time part of `make-app` is the S combinator and `compile-term` just supplies the initial compile-time environment.

3.6 Pass free variables

A refinement of the multi-argument approach of Sec. 3.4 passes only the values of the free variables at run time. There is no need to maintain an environment at compile time, instead compilation generates the list of free variables along with

```
(define (make-var x)
  (list (list x)
        (lambda (x) x)))

(define (make-lam x e)
  (let* ((fv-e (freevars e))
         (comp-e (compiled e))
         (fv (set-subtract fv-e x))
         (comp ((get-lambda* (length fv)
                             1
                             (names->index (append fv (list x)) fv-e))
                comp-e)))
    (list fv comp)))
```

Fig. 5. Combinators to pass free variables

the code and the code expects the values of the variables to be passed to it in the sequence specified by the list. Hence, the active syntax type is a dependent sum $\sum \texttt{env} : \texttt{Var}^*.(\texttt{Val}^{|\texttt{env}|} \to \texttt{Val})$, i.e., a pair containing the list of the free variables and the corresponding function. The functions `freevars` and `compiled` are the projections on the first and second component.

Due to the invariant that exactly the values of the free variables are passed to each expression the compilation of variables is straightforward (see Fig. 5).

Compilation of lambda abstraction is more involved, due to the fact that the abstracted variable may not occur free in the body. It relies on a cached function `get-lambda*` that takes the number of variables to be expected from the context (`(length fv)`), the number of variables abstracted (`1`), and a list of numbers that select those variables that are passed on to the body (`(names->index (append fv (list x)) fv-e)`).

For applications, there is similar cached function `get-application*` that selects and distributes the values of the free variables.

3.7 Shallow binding

Shallow binding [4] inspires an approach which represents variables by reference cells. It is safe to bind variables to fixed mutable cells *at compile time*, provided that each cell contains a stack of values with the current value on top.

The type of the active syntax is $\texttt{CEnv} \to \texttt{Unit} \to \texttt{Val}$. There is no explicit run-time environment. All name resolution is performed at compile time.

To conserve space, we only consider the compilation of a lambda abstraction in Fig. 6. The compile-time environment `env` is a list of pairs of variable names and cells. Compilation creates a new cell `c0` corresponding to the variable `v0`. Next, it collects the cells that make up the current environment in `cells`. Finally, it compiles the body of the lambda while binding `v0` to `c0`.

The resulting thunk which is executed at run time forms a closure consisting of the current values of the cells in the environment and returns the real lambda

```
(define (make-lam v0 e)
  (lambda (env)
    (let* ((c0 (make-cell '()))
           (cells (map cdr env))
           (body (e (cons (cons v0 c0) env))))
      (lambda ()
        (let ((freevals (map (lambda (c) (car (cell-ref c))) cells)))
          (lambda (x0)
            (cell-set! c0 (cons x0 (cell-ref c0)))
            (for-each (lambda (c x) (cell-set! c (cons x (cell-ref c))))
                      cells freevals)
            (let ((result (body)))
              (cell-set! c0 (cdr (cell-ref c0)))
              (for-each (lambda (c) (cell-set! c (cdr (cell-ref c))))
                        cells)
              result)))))))
```

Fig. 6. Compilation of lambda abstraction using references

`(lambda (x0) ...)`. Applying the lambda establishes the new binding of the variable `v0` and installs the values of the free variables from the closure. After running the body, it restores all bindings to their previous state before returning the result.

The compilation of a variable obtains its associated cell and returns a thunk that returns the top of the stack stored in that cell.

The compilation of an application wraps the application in a thunk that runs the thunks of the subexpressions as appropriate.

On entry to the body of a lambda only the cells corresponding to free variables of the body need updating. This optimization is straightforward.

4 Results

This section reports practical experiments with implementations of most of the approaches described. We have run three sets of benchmark programs on two different interpreted Scheme implementations, Scheme48 version 0.51 and Gambit 3.0. Scheme48 compiles to byte code for subsequent interpretation whereas Gambit is a quasi source-level interpreter. All measurements were performed on a 233MHz Pentium II machine with 256MB of memory, running FreeBSD 2.2.5. We report the average time of ten runs of the same computations, using the respective `time` commands of the Scheme systems.

We have measured the following variations of code splicing:

scheme section 3.1, generate Scheme source and compile using `eval`;
ski-opt section 3.2 but using an intermediate combinator expression;
ski-comp is the fully deforested version;
flat section 3.4, pass the run-time environment as a flat tuple;

linked section 3.5, implement the environment by a linked list;
free section 3.6, flat environment restricted to the free variables;
ref section 3.7, naive shallow binding; **ref-free** only updates free variables.

We have measured times for

constr construction: generate an object of type `ASyntax`;
comp compile with `compile-term`;
run run the resulting procedure.

We have used TDPE to generate lambda expressions as follows:

church/n constructs the text of the Church numeral for n (see Table 1);
tiny specialize the Tiny (a little imperative language [18]) interpreter wrt. a factorial program (see Table 2);
mixwell specialize the Mixwell (a first-order functional language [14]) interpreter wrt. a program with about 300 functions (see Table 3).

In almost all cases, the construction time of the active syntax is equal to the construction time of the corresponding source. The exception is **free** which performs a free variable analysis at construction time. This amounts to a slowdown by a factor of 10 for **church** and 4 for **tiny** and **mixwell**.

With Scheme48, the time taken for construction and splicing is linear in the size of the source term. In Gambit, it seems to be quadratic. Compilation using `eval` seems to take quadratic time for both systems. In the **church** benchmark, Scheme48 ran out of memory for **scheme/10000** given the maximum possible heap size `-h 33539072`.

The holes in the **mixwell** table come from heap overflow for **linked** and an implementation restriction for **scheme**: the distributed version limits the nesting depth of bindings to 256 while the **mixwell** program has a nesting depth of well over 300. The table demonstrates that code splicing overcomes such restrictions.[1]

Overall, the **free** approach comes out fastest for all benchmarks. However, its compilation time is much slower than all others. The second choice is between **linked** and **ref-free** which combine extremely fast compilation time with fairly good execution times. The **church** benchmark seems to give a fairly distorted picture because of its uncharacteristic behavior for **ref-free**. The **tiny** and **mixwell** programs appear to yield more realistic results.

In terms of compilation time of **ski-comp** vs. **ski-opt**, deforestation pays off: **ski-comp** is two times faster. In terms of run time, **ski-comp** is slightly slower because **ski-opt** avoids some uses of the I combinator by inspecting the text of the combinator expression; this is impossible for **ski-comp** which does not generate an intermediate result.

For the **tiny** and **mixwell** benchmarks, it is interesting to compare with the run time of the original interpreter prior to specialization with TDPE. For **tiny**, its run time is 0.47 seconds for Scheme48 and 0.165 seconds for the Gambit interpreter. For **mixwell**, it is 0.712 seconds (Scheme48) and 0.64 seconds

[1] This restriction has been removed in the mean time.

	Scheme48			Gambit Interpreter		
technique	constr	comp	run	constr	comp	run
scheme/100	0.00	0.28	0.00	0.005	0.003	0.001
scheme/1000	0.03	23.08	0.00	0.085	0.069	0.005
scheme/10000	0.37	—	—	2.640	3.510	2.776
scheme/100000	3.75	—	—	206.993	352.020	91.521
ski-opt/1000	0.03	0.22	0.01	0.073	0.477	0.703
ski-opt/10000	0.37	2.26	0.17	2.649	9.535	17.414
ski-opt/100000	3.72	23.65	1.71	191.800	547.981	740.677
ski-comp/1000	0.03	0.10	0.01	0.087	0.370	0.684
ski-comp/10000	0.37	1.14	0.19	2.676	6.174	21.829
ski-comp/100000	3.74	11.66	1.95	204.936	276.527	868.653
flat/1000	0.03	0.03	0.01	0.087	0.066	0.352
flat/10000	0.37	0.32	0.17	3.649	3.020	8.827
flat/100000	3.71	3.28	1.76	206.176	280.349	450.799
linked/1000	0.04	0.13	0.02	0.090	0.126	0.888
linked/10000	0.40	1.33	0.25	3.099	4.161	13.969
linked/100000	4.08	13.47	2.43	201.652	354.373	445.056
free/1000	0.32	0.00	0.00	0.394	0.000	0.349
free/10000	3.24	0.00	0.08	7.775	0.000	6.677
free/100000	32.55	0.00	0.80	240.686	0.000	125.246
ref/1000	0.03	0.03	0.00	0.086	0.061	0.344
ref/10000	0.36	0.39	0.07	3.909	2.841	9.201
ref/100000	3.68	3.96	0.79	214.278	275.319	385.260
ref-free/1000	0.03	0.07	0.00	0.161	0.117	0.355
ref-free/10000	0.37	0.73	0.08	4.695	3.886	10.433
ref-free/100000	3.74	7.50	0.87	214.103	322.110	604.603

Table 1. Run times for Church numerals (in sec)

	Scheme48				Gambit Interpreter			
technique	constr	comp	run	ratio	constr	comp	run	ratio
linked	0.01	0.01	0.52	1.13	0.011	0.008	0.275	1.666
flat	0.01	0.00	0.72	1.55	0.011	0.001	0.523	3.169
free	0.04	0.00	0.43	0.94	0.030	0.000	0.182	1.103
ref-free	0.01	0.00	0.72	1.53	0.010	0.009	0.534	3.236
ref	0.01	0.00	1.03	2.26	0.012	0.002	0.875	5.303
scheme	0.01	0.04	0.40	0.85	0.013	0.003	0.039	0.236

Table 2. Timings for the Tiny interpreter (sec)

	Scheme48				Gambit Interpreter			
technique	constr	comp	run	ratio	constr	comp	run	ratio
linked	3.45	—	—	—	3.870	32.329	1.158	1.89
flat	3.38	4.36	2.06	2.90	3.841	0.852	2.407	3.70
free	13.44	0.00	0.14	0.20	7.826	0.000	0.093	0.15
ref	3.33	6.42	3.80	5.34	3.743	1.582	3.966	6.26
ref-free	3.35	6.42	0.32	0.44	3.738	2.858	0.843	1.37
scheme	3.42	11.88	—	—	3.723	2.966	0.036	0.06

Table 3. Timings for the Mixwell interpreter (sec)

(Gambit interpreter). The *ratio* column contains the run time of the compiled version divided by the run time of the fully interpreted version.

Both experiments require multi-argument lambda abstraction and application. The corresponding active syntax constructors are not implemented for **ski-opt** and **ski-comp**, hence there are no results for these techniques.

The very first runs of **linked** and **free** are about an order of magnitude slower than the rest because they fill the caches.

Assessment Obviously, our mileage varies depending on the system that we use. For the Scheme48 system, which compiles to byte code, the code splicing approach seems to be viable and the **free** and **ref-free** implementations give encouraging results.

For the Gambit interpreter, the implementation of `eval` is blindingly fast and gives very good results, so this is the method of choice for the Gambit system. Why? Gambit itself uses a code splicing strategy to implement `eval` [9], but since the Gambit interpreter itself is compiled `eval` splices fully compiled code. In contrast, our experiments were conducted with the interpreter, only. Therefore, we expect to obtain better results when we use Gambit's compiler because in that case our combinators are fully compiled, too.

5 Related work

Writing staged interpreters has become popular since Feeley's thesis [8, 9]. It is now a standard technique that is taught in introductory textbooks [1]. The partial evaluation community also exploits this style and a set of transformations to improve the staging properties to specialize programs and generate compilers efficiently [13]. Holst and Gomard [12] show that the same style and very similar transformations enable a lazy functional programming language to achieve similar specialization effects as partial evaluators. In contrast, we are considering staging for strict functional programs.

One of our sources of inspiration is Augustsson's ingenious implementation of lmli, the interactive part of the lazy ML compiler [3]. Lmli compiles an interactive definition to an (SKI) combinator expression and maps it into executable

code by folding the expression with respect to the compiled definitions for the combinators (with type checking turned off). This results in respectable speed for code typed in from the terminal and at the same time trivial interfacing with compiled code from other modules and with the run-time system, exactly the goals of our setting. Our design space is somewhat less constrained than Augustsson's: Since we are using an untyped, strict, and impure language, there are interesting options to consider besides SKI combinators.

Two works deal with compiling specialized source to byte code on-the-fly. Sperber and Thiemann [19] implement a back end for a traditional partial evaluator. As in the present work, they reinterpret the active syntax, but their implementation generates byte code on-the-fly. Balat and Danvy [5] construct an internal representation of the source term which they submit in toto to the byte code compiler. They do not deforest the intermediate result. Like the present work they implement a back end for TDPE, thus side-stepping problems with top-level definitions and primitive operations that contributed to the complexity of the other work [19].

6 Conclusion

We have investigated part of the design space for compilation by code splicing. We avoid an expensive compilation step by using higher-order functions to splice together precompiled pieces of code at run time. Each of the techniques that we propose has counterparts in the implementation of functional programming languages. Similar to the tradeoffs between the different implementation techniques, the splicing techniques have tradeoffs that depend on the particular source programs and on the underlying implementation.

Some of the methods have not been considered before in this context, most notably the compositional compilation to SKI combinators, director strings, and the strategy that employs shallow binding.

References

1. Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass., second edition, 1996.
2. Ali-Reza Adl-Tabatabei, Michal Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast, effective code generation in a just-in-time Java compiler. In Keith D. Cooper, editor, *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 33 of *SIGPLAN Notices*, pages 280–290, Montreal, Canada, June 1998. ACM.
3. Lennart Augustsson. The interactive lazy ML system. *Journal of Functional Programming*, 3(1):77–92, January 1993.
4. Henry G. Baker, Jr. Shallow binding in Lisp 1.5. *Communications of the ACM*, 21(7):565–569, July 1978.

5. Vincent Balat and Olivier Danvy. Strong normalization by type-directed partial evaluation and run-time code generation. In *Proceedings of the ACM SIGPLAN Workshop on Types in Compilation (TIC'98)*, volume 1473 of *Lecture Notes in Computer Science*, Kyoto, Japan, March 1998.
6. Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. The categorical abstract machine. In *Proc. Functional Programming Languages and Computer Architecture 1985*, volume 201 of *Lecture Notes in Computer Science*, pages 50–64. Springer-Verlag, 1985.
7. Olivier Danvy. Type-directed partial evaluation. In *Proc. 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 242–257, St. Petersburg, Fla., January 1996. ACM Press.
8. Marc Feeley. Deux approches ý l'implantation du language Scheme. Master's thesis, UniversitÈ de MontrÈal, 1986.
9. Marc Feeley and Guy Lapalme. Using closures for code generation. *Computer Languages*, 12(1):47–66, 1987.
10. Anthony J. Field and Peter G. Harrison. *Functional Programming*. Addison-Wesley, 1988.
11. Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. MIT Press and McGraw-Hill, 1992.
12. Carsten Kehler Holst and Carsten Krough Gomard. Partial evaluation is fuller laziness. In Paul Hudak and Neil D. Jones, editors, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '91*, pages 223–233, New Haven, CT, June 1991. ACM. SIGPLAN Notices 26(9).
13. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
14. Neil D. Jones, Peter Sestoft, and Harald S¯ndergaard. An experiment in partial evaluation: The generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications*, pages 124–140, Dijon, France, 1985. Springer-Verlag. LNCS 202.
15. Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *SIGPLAN Notices*, 33(9):26–76, September 1998.
16. J. R. Kennaway and M. Ronan Sleep. Director strings as combinators. *ACM Transactions on Programming Languages and Systems*, 10(4):602–626, 1988.
17. Peter Lee. What is run-time code generation good for? Panel contribution at APPSEM meeting, September 1998. Pisa, Italy.
18. Larry C. Paulson. Compiler generation from denotational semantics. In Bernhard Lorho, editor, *Methods and Tools for Compiler Construction*, pages 219–250. Cambridge University Press, 1984.
19. Michael Sperber and Peter Thiemann. Two for the price of one: Composing partial evaluation and compilation. In *Proc. of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 215–225, Las Vegas, NV, USA, June 1997. ACM Press.
20. Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In Charles Consel, editor, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '97*, pages 203–217, Amsterdam, The Netherlands, June 1997. ACM Press.
21. David A. Turner. *Aspects of the Implementation of Programming Languages*. PhD thesis, University of Oxford, 1981.
22. Philip L. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990.

Expressing Structural Properties as Language Constructs*

Shriram Krishnamurthi, Yan-David Erlich, and Matthias Felleisen

Department of Computer Science
Rice University
Houston, Texas 77005-1892, USA

Abstract. A program should document its organization and decisions about the programming process. Since the programmer's thinking about programming and program organization continually evolves, languages inevitably prove unable to state these decisions in a precise and adequate fashion. Macro systems could provide a convenient way to extend a language with such statements, if they had more structure than traditional C- and Lisp-style macros provide.
With our system, McMicMac, designers can express a variety of specifications as language constructs, including program representations of design patterns, high-level recursive programming operators, and collaboration-based design mechanisms. Unlike traditional macro systems, McMicMac offers a simple yet powerful means for describing specifications, prevents unintentional name clashes, provides feedback in terms of the programmer's source, and has modular mechanisms for managing specifications. We have implemented and used McMicMac to define several groups of extensions.

1 Introduction

A program specifies not only behavior, but also decisions concerning its organization and about the programming process. It expresses this information in terms of the constructs of the programming language in which it is written. Any language, however, has only a limited set of constructs, while programmers constantly invent new concepts and grow their vocabulary about both the program and the programming process. This leads to a gap in expressiveness.

Many programming methodologies try to narrow this gap. Program design patterns [12] that capture some details of the structure of the program are one example; designs based on collaborations [3] are another. Yet other examples are abstract structure operators [22] and adaptive programming specifications [20]. They permit programmers to describe a program's organization and development, and can also be mapped into constructs in the underlying programming language.

* This research was partially supported by NSF grants CCR-9619756 and CCR-9708957, and by a Texas ATP grant.

S.D. Swierstra (Ed.): ESOP/ETAPS'99, LNCS 1576, pp. 258–272, 1999.
© Springer-Verlag Berlin Heidelberg 1999

These specifications are used in various ways. They are typically described only in comments, translated into code by hand, or embedded in the program as linguistic extensions that are processed by special-purpose tools. Comments are unchecked and therefore unreliable. Implementing specifications manually is painstaking, deprives program maintainers of crucial information, and makes it difficult to ensure that constantly-evolving programs are meeting the intended specifications. In contrast to these, special-purpose tools can both verify that the program satisfies the specification and translate the specifications into instructions in the underlying programming language.

Unfortunately, these special-purpose tools take considerable effort to implement. Each tool must parse the complete programming language and its own extensions, and implement its own rewriting strategy. Also, since these tools hard-wire their understanding of the underlying language, they may not be able to process a program extended with constructs from some other specification process. Finally, extending such tools may be cumbersome, and can differ considerably between tools.

We propose an alternate approach to expressing such structural properties. Our approach, based loosely on Scheme macros, offers more structure and more facilities than traditional C or Lisp-style macro expanders. It simplifies the task of specifying properties, and tries to preserve their integrity when communicating information between programming tools and the programmer. Finally, by providing a common framework, it allows different specifications to interact atop the same language and with each other.

The rest of this paper is organized as follows. Section 2 discusses some of the structural properties that programmers use, and how they can be implemented. Section 3 describes features that would be useful to implement the specifications of Section 2, and how our implementation supports them. Section 4 surveys the literature of related work. The last section summarizes the ideas in this paper, and presents some directions for future work.

2 Programming with Specifications

In this section we present some examples of structural properties and discuss how to integrate them into the programming process.

2.1 Software Patterns

Design patterns sketch a solution to a class of related problems. By customizing a pattern to a given context, a programmer can reuse the embodied solution. Some patterns represent general "architectures" with respect to a programming language that can be instantiated effectively in different contexts. We call these architectures *software patterns*.

Figure 1 illustrates a sample use of a software pattern[1] in Java. The code in the figure uses the Adapter pattern, which enables the reuse of an existing

[1] All the patterns mentioned in this paper are taken from Gamma, et al. [12].

```
interface OutputRoutines {
   void drawLine ();
   void printText (); }
class VendorGraphics {
   void drawSolidLine () {};
   void renderText () {}; }
// GraphicalOutput adapts VendorGraphics to OutputRoutines
class GraphicalOutput implements OutputRoutines {
   VendorGraphics lowLevelDriver;
   public void drawLine () { lowLevelDriver.drawSolidLine (); }
   public void printText () { lowLevelDriver.renderText (); } }
```

Fig. 1. Adapter Pattern: Java Version

```
(interface OutputRoutines
  (methods (void drawLine ())
           (void printText ())))
(class VendorGraphics
  (methods (void drawSolidLine ())
           (void renderText ())))
(Adapter GraphicalOutput adapts VendorGraphics
                         to OutputRoutines
                         as lowLevelDriver
  (fields)
  (methods
    (public void drawLine () (lowLevelDriver . drawSolidLine ()))
    (public void printText () (lowLevelDriver . renderText ()))))
```

Fig. 2. Adapter Pattern: Specification-Based Version

class with a different interface. More specifically, suppose an existing class implements the desired functionality but does not implement the desired interface. An Adapter acts as a surrogate for this class by implementing the desired interface, forwarding requests to instances of the existing class and tailoring responses to its interface.

In our example, *OutputRoutines* is an interface that represents output generators, and is implemented by actual generators on various devices. The vendor-provided *VendorGraphics* performs graphical output, but *VendorGraphics* does not implement *OutputRoutines*, and its interface is slightly different. In Figure 1, the programmer creates an Adapter class, *GraphicalOutput*, that forwards requests to an instance of *VendorGraphics*. (For brevity, we elide the details of the actual methods.) The relationship is documented informally through the comment.

The programmer could now make the mistake of defining the class *Client* with the type of some variable *var* to be *VendorGraphics* but assigned an instance

of *GraphicalOutput*. Unfortunately, *VendorGraphics* is the class being adapted rather than the interface it is being adapted to:

```
class Client {
    VendorGraphics var = new GraphicalOutput ();
}
```

The type-checker flags this assignment with the following error message:

```
test.java:13: Incompatible type for new. Can't convert
  GraphicalOutput to VendorGraphics.
   VendorGraphics var = new GraphicalOutput ();
                        ^
```

This example illustrates several effects of programming directly in terms of a pattern's constituent constructs:

- The implementation of the Adapter pattern in terms of plain classes and interfaces obscures the pattern's identity, and thus decreases the clarity of the program. This makes it more difficult for readers to understand the structure and intent of the code.
- It increases the potential for errors due to the volume of code. Pattern code handles administrative tasks such as maintaining invariants, which a programmer must correctly implement during the initial development and remember to update during maintenance. Even in this simple example, for instance, the programmer must remember to make *GraphicalOutput* implement the interface *OutputRoutines*, and declare the type of *lowLevelDriver* as *VendorGraphics*.
- It becomes difficult to replace the code for the pattern instance with an improved version, e.g., one that is more extensible or offers better protection against errors.
- All feedback is reported in terms of the individual units of code, and the programmer must then manually extrapolate from the error message to the original pattern-based design.

In short, programming directly with the constituents of patterns can potentially affect programmers at every stage: implementation, debugging and maintenance.

Figure 2 shows the same program (translated into a parenthesized version of Java), except it makes the Adapter explicit by using an **Adapter** construct, which is translated into the equivalent of the code in Figure 1. The use of an Adapter makes it clear that there is a mismatch between *OutputRoutines* and *VendorGraphics*, and highlights how this can be overcome. The equivalent of *Client*'s Java declaration is then

```
(class Client
  (fields (VendorGraphics var = (new GraphicalOutput ()))))
```

This results in the error message

```
test.cj:15.33-15.56: Incompatible assignment type. Can't convert
  GraphicalOutput to VendorGraphics.
GraphicalOutput created by Adapter at test.cj:7.2-7.8.
```

which uses the Adapter declaration to report the error in terms of the programmer's pattern code, not just the expanded constructs. The error message frees the programmer from having to track manually what each construct can introduce, which can be especially difficult since the expansion of the construct can change over time. This error message is produced by a generic type-checker for our parenthesized version of Java; Section 3.3 describes how this information is generated in greater detail.

2.2 Composition Invariants

Smaragdakis and Batory [24] present several approaches to validating the correctness of module compositions. They express the compositional requirements through the type system. For instance, suppose the implementation of a virtual machine could include a module that tags the memory representations of data (*Tag*) and another that implements garbage collection (*GC*). Furthermore, suppose that the *GC* module requires *Tag* to be present in the module composition. To ensure that the *Tag* module has already been added, *GC* declares a dummy variable of type *TagIncludedProperty*, which is the name of an empty class declared only by the *Tag* module. Thus, if *Tag* is not included in the composition hierarchy before *GC*, the dummy variable declaration will raise a type error, which indicates an erroneous composition.

Implementing this approach via explicit constructs for validating compositions offers many advantages:

1. These class and variable declarations play no role in the execution of the program. Therefore, they obscure its behavior, potentially making it more difficult to maintain.
2. Some of these invariants are tricky to implement. It is thus safer to have them implemented automatically by a program than manually by the programmer. This is especially important since some negative properties require a quadratic number (in the size of the program and the number of properties) of dummy variable declarations. Also, when a new property is introduced, the programmer has to add it to each class by hand. With a properly designed language extension, the programmer can instead establish a single point of control and greatly reduce the work required to specify the program's properties.
3. When an invariant is violated, the programmer should get an error in terms of the actual invariant, not just in terms of the (often mangled) names used to represent it in the source. Smaragdakis and Batory identify this issue as one of the major problems in their approach [24, pg. 562].
4. While adding the dummy variable declarations, a programmer might accidentally choose a name that is already in use for a different purpose.

2.3 Other Specifications

Several other specification techniques can be handled with our proposed solution. For instance, abstract structure operators [22] are used to simplify the process of describing traversals over recursively-defined types. They are generated automatically from the type specification. Instead of defining recursive procedures over these types, programmers use and compose these high-level operators, which are then translated into regular recursive procedures. The translation again raises the issues of name-clash management and feedback-reporting, which can be handled by our proposal.

3 Programming Support for Specifications

In the preceding section, we have mentioned several desirable properties for a programming tool that supports specifications. To summarize, such a tool

1. should make it easy to describe the syntax of the property;
2. should permit a simple specification of the translation into code fragments for those properties that require an implementation;
3. must help the designer avoid name capture problems;
4. should provide feedback to programmers in terms of the original specifications, not just their expansions; and,
5. should support the definition of disjoint groups of specifications.

We have developed a language elaborator, McMicMac, that meets these requirements. In this section, we describe the key properties of McMicMac, and explain how they address the aforementioned needs. For information concerning McMicMac's implementation, we refer the reader to the technical report [18].

3.1 Describing Specifications

Many specifications have operational meanings. For such specifications, *designers must be able to specify the transformation that maps an instance of a specification into constructs in the underlying language*. The mechanism must be both convenient enough to simplify the addition of new transformations, and powerful enough to specify potentially complex ones.

A software pattern, for example, can be thought of as a parametric body of code that is specialized at each particular use. To use a pattern, the programmer must provide code fragments for the parameters. The pattern implementation assembles a program component by splicing the code fragments into the parameter positions of the body of code. Thus the pattern designer needs a tool that (1) permits the definition of parameterized bodies of code, and (2) generates code from arguments supplied for the parameters. McMicMac supports these operations felicitously through shape matching,[2] a mechanism due to Kohlbecker and Wand [17].

[2] This is traditionally known as "pattern matching", but we adopt different terminology to avoid confusion with software patterns.

```
(define-macro Adapter
  (rewrite
    (Adapter adapterName adapts adapteeType
                          to desiredInterface
                          as adapteeVariable
      (fields field-decls ...)
      (methods method-decls ...))
    (with-keywords adapts to as fields methods)
    (as
      (class adapterName implements desiredInterface
        (fields (adapteeType adapteeVariable)
                field-decls ...)
        (methods method-decls ...)))))
```

Fig. 3. Adapter Pattern Definition

Shapes are built from keywords, shape variables, and sequences. The shape-matcher works by comparing each phrase in a program against the collection of defined shapes. A phrase matches a shape if the phrase uses keywords and sequences in the same manner as the shape. In this case, the shape matcher binds the corresponding parts of the inputs to the shape variables in the template. It then generates an output phrase based on the corresponding macro definition, which yields a new phrase. It expands this phrase again, continuing until the phrase does not match any defined shape. Figure 3 presents the definition of the Adapter pattern, which is used in Figure 2. This use results in code that is equivalent to that of Figure 1.

The matcher treats ellipses (...) specially. An ellipsis follows a ("head") shape in a sequence,[3] and matches a source sequence of zero or more instances of the head shape. It binds each shape variable in the head shape to a sequence. This sequence consists of the sub-terms, in order, of the terms in the source sequence that correspond to the shape variable's position in the head shape. Ellipses can be nested to arbitrary depth. Each nesting level introduces a nested sequence in the binding of a shape variable.

The definition and application of the Adapter pattern illustrate the use of ellipses. One sub-shape of the Adapter is (**methods** *method-decls* ...). In the example, there are two methods in the shape:

```
(methods
  (public void drawLine () (lowLevelDriver . drawSolidLine ()))
  (public void printText () (lowLevelDriver . renderText ())))
```

The shape-matcher binds *body-terms* to a sequence of two pieces of code:

```
(public void drawLine () (lowLevelDriver . drawSolidLine ()))
```

[3] The head of a shape cannot be ..., else the shape is considered ill-formed.

and

(**public void** *printText* () (*lowLevelDriver* . *renderText* ()))

As Figure 3 suggests, the shape-matcher not only verifies that the specification has a proper form, it also generates output. Put differently, adding a specification via McMicMac assigns a precise meaning to a specification. The shape-generator works by essentially inverting the matching process. In particular, it too can handle nested ellipses; each shape variable must appear under as many ellipses in the output shape template as it did in the input template.

Ellipses play an important role not only in laying out the notation, but also in defining the meaning of specifications. They elucidate the inductive structure of the specifications, an aspect that is often overlooked in pattern definitions. Specification implementors use ellipses wherever they want to leave the number of concrete entities unspecified and let the programmer provide this information. In the Visitor pattern, for instance, the programmer's specification not only generates the visited type but also the visiting methods that characterize the pattern. Thus ellipses capture what Lauder and Kent term "purity" [19]: they reflect *all* instances of the pattern, not just a single deployment from which the programmer must extrapolate to the general case.

3.2 Avoiding Inadvertent Interference in Code

When a specification expands into code that co-mingles with code written by the programmer, one code fragment might inadvertently bind or use a name that is bound or used in the other fragment. Since neither the programmer nor specification designer can foresee all such situations, *the elaborator must ensure that the programmer's and the specification's code do not inadvertently interfere with the lexical properties of each other.*

Macro expansion can ensure that variable bindings and uses in user code do not bind and are not accidentally bound by bindings and uses in generated code, and vice versa. This process is called *hygienic* expansion [16]. In McMicMac, the hygienic macro expander works by marking each step of expansion with a distinct "timestamp", and accumulating timestamps on every term. To determine whether a variable is bound, the expander considers both the name and timestamps on a variable, and identifies only those variables with the same name and timestamps. Since code from the user and from each macro elaboration have different timestamps, this prevents inadvertent capture of variables.

For example, consider Smaragdakis and Batory's method for verifying modular compositions [24]. Their specification mechanism introduces declarations of the following form (in C++, using our example from Section 2.2):

template < **class** *Super* > **class** *Tag* : **public** *Super* {
  protected:
    class *TagIncludedProperty* {};
  . . . }

is the *Tag* module, and

```
template < class Super > class GC : public Super {
  private:
    TagIncludedProperty dummy1;
  ... }
```

is *GC*. The variable *dummy1* here has two potential problems:

- It interferes with any other attempt to declare a variable of the same name. Such a variable may be declared not only by the user, but also by some other specification mechanism. Even if the specification designer chooses an obscure name, some specifications can be applied multiple times, so one instance of the specification can clash with another one.
- Some other part of the class's body may accidentally mention the name *dummy1*. This should produce an unbound variable error, but because of the above declaration, it may produce an unexpected type error or, worse, run without error and compute an incorrect answer.

Hygienic expansion ensures that the identifier *dummy1* that is introduced by the macro does not conflict with any other identifiers, including those named *dummy1* but coming from other sources (including other uses of the same macro). This is essential when a program is developed by a team of programmers, who would otherwise find it extremely difficult to prevent inadvertent name collisions.

3.3 Maintaining the Integrity of Specifications in Feedback

A programming environment includes numerous tools that provide feedback about the program. These include type-checkers, program slicers, debuggers, profilers, and so forth. In all these cases, the analyzed program includes both code written by the programmer and code generated by a specification. Even when these are co-mingled, *feedback must be in terms of the source written by the programmer, not just in terms of the generated code.* This is essential for two important reasons:

- to maintain the impression that the programmer is coding at the level of the specification, not at the level of the code it generates; and,
- because some of the code in question may not appear in the source text.

Feedback of this form is particularly important when the juxtaposition of user and specification code results in complex errors.

McMicMac provides an interface for tools that wish to report feedback. The interface consists of two kinds of information, and protocols that process this information. The two kinds of information are:

source-correlation For each term, McMicMac notes the source location of the phrase that generated it. This information is maintained through the macro expansion process. Thus when a tool needs to provide feedback about a term,

it can report it in terms of the source phrase that the programmer needs to examine. This is especially helpful to graphical interfaces, since they can highlight the appropriate source text.

elaboration-tracking McMicMac also maintains a history of transformations applied to each elaborated term. In the example of Section 2.1, it records that *GraphicalOutput* is generated by **Adapter** (with its source location). In more complex examples, one specification might expand into the use of another. In these cases, conventional error messages in terms of the constituents might be especially unhelpful.

The tools themselves do not need to process this information. For example, the McMicMac error checker processes only the ClassicJava language, and has no knowledge of software patterns. Instead, the information is extracted and processed by a McMicMac protocol, which generates feedback of the sort shown in Section 2.1.

3.4 Organizing Specifications by Layers

Specification designers need modular constructs to organize collections of specifications. McMicMac provides a module mechanism called *vocabularies*. Each specification must be placed in some vocabulary; a vocabulary represents a language fragment, derived by combining the individual macros in that vocabulary. Vocabularies can be combined with other vocabularies. In particular, a vocabulary representing a specification can be combined with different base languages. These base languages might vary in at least two different ways:

1. By layering several vocabularies atop a base language, the programmer can combine multiple specification schemes. This may be impossible to achieve when each specification method provides its own tool, since these tools may reject or incorrectly process the other forms of specification.
2. Users can customize how they use the specification method by picking different base languages. For instance, a sophisticated programmer will want a powerful base language, while teachers might prefer a simpler one for their students.

A vocabulary also represents one way in which a specification analyzes a program; some specifications may process the same program fragment several times. For instance, many common program analyses and optimizations can be partitioned into "collection" and "modification" phases. The first phase traverses the program and collects information relevant to the analysis. This information is processed, and then used by the second phase, which rewrites the program appropriately. Each of these phases can be elegantly encoded as a vocabulary.

3.5 Implementation Status

McMicMac is currently implemented in MzScheme [9]. We have constructed vocabularies for several versions of Scheme [8] and one for ClassicJava [10], an

idealized, parenthesized subset of Java. We have enhanced these base languages with several structural transformers, including software patterns corresponding to each of the design patterns described in the catalog of Gamma, et al. [12], abstract structure operators [22], and execution profilers [1].

The facilities of McMicMac helped us implement all these tools in short order. Our implementation produces information that helps us generate useful and intuitive feedback for the programmer, such as the error messages shown in this paper. Though the current implementation of McMicMac is restricted to parenthesized syntax, it can be extended to more traditional syntaxes along the lines of Cardelli, et al.'s work [6].

4 Related Work

Researchers have studied various specification mechanisms. We partition these into several categories below, though naturally there is some overlap.

4.1 Software Patterns

With the increasing popularity of software patterns, many researchers have proposed ways to design tools and languages that support them.

Graphical and Translation Tools. Patterns are frequently described using graphical notations. Researchers have thus tried to design graphical tools that support programming with patterns. Budinsky, et al. [5] describe a GUI-based package which lets users prepare a diagrammatic representation of the pattern-level layout of their program, and then fill the components with source code. The system's back-end uses a script to generate code from the user's input. Florijn, et al. [11] have built browsers to support their work on fragments, described below. Meijler, et al. [21] have built a modeling environment called FACE in which programmers use patterns by cutting-and-pasting from a library of patterns in UML, and then filling in the blanks with code. The FACE environment is unusual in that it supports patterns throughout the program's life, not just during its initial creation.

Graphical tools are attractive because they can provide an intuitive interface to the programmer, but they have several shortcomings.

1. They often do not clarify the distinction between a pattern, which can have a potentially unbounded number of components, and a pattern instance, which has a fixed, finite number. Often, they provide a template with a small number of placeholders for concrete entities, which the user must duplicate by hand to obtain the desired number; this process is both laborious and error-prone.
2. Some contain an analysis engine for the pattern language, but none of them interact with the underlying programming language to provide an equally convenient interface for feedback once the patterns have been translated into code. This could be remedied by using a system such as ours as a back-end for the graphical tools.

3. The systems descriptions do not clarify how groups of patterns can be organized, or how to extend tools to accommodate other specification notations.

Soukup [25] proposes implementing patterns through C-style macros. These macros lack inductive constructs, offer no facilities for reporting feedback, are not hygienic, and have no modular structure. Hedin [13] defines patterns in terms of attribute grammars. His work currently uses these to reconstruct patterns in existing source code.

Constraint Models. Both Bosch [4] and Florijn, et al. [11] describe interesting approaches to pattern-based programming. Bosch takes a top-down view on programming with patterns. His system uses a layered object model to describe constraints on entities that participate in patterns. His paper also lays out four significant problems in implementing design patterns, of which our work addresses three: *traceability*, *reusability* and *implementation overhead*.

Programmers who use Florijn, et al.'s system write "fragments", which are elements of programs and patterns. They then designate fragments to play roles in instances of patterns. The pattern designer specifies constraints (similar to contracts [14]) that judge when a collection of fragments satisfies the pattern. This approach gives the programmer the flexibility of restructuring the system easily. It is, however, unclear how their system handles feedback across elaboration or the inductive structure of patterns.

4.2 Other Specification Implementations

Adaptive programs [20] consist of declarations of class hierarchies, traversals, and visitors. The traversal maps the order in which to visit nodes in the hierarchy, while the visitors determine what actions to perform at each node. Adaptive programs are compiled into programs in a conventional language like Java using the Demeter tool [20]. Though it uses sophisticated algorithms to compile specifications into efficient code, Demeter does not report feedback across the elaboration process.

Aspect-Oriented Programming [15] is implemented by *weavers*, which combine the original program with code derived from the aspect specifications. The implementation for Java, called AspectJ, produces generic Java code that can be compiled by any generic compiler. As a result, however, it does not offer the user support at the level of aspect specifications. Instead, the primer [27] instructs users to track errors based on their file location.

Batory and Geraci [2] describe a process for checking that code generated from specifications meets compositional requirements. Their work imposes a type system on a program's components, and uses this system to verify the validity of compositions. When a composition is invalid, the system can suggest reasons for the error. Their work can be used to improve the error reporting for modular, collaborative designs [24].

4.3 Macro Systems and Extensible Grammars

McMicMac is a close relative (and descendent) of other language extension mechanisms, including macro systems and extensible grammars. The derivation and use of McMicMac's shape-matching mechanism was described by Kohlbecker and Wand [17]. The design of hygienic macro expanders is due to Kohlbecker, et al. [16], though our algorithm is derived from that of Dybvig, et al. [7]. The latter work also outlines a method for maintaining source-correlation.

Like a macro system, Taha and Sheard's MetaML [26] gives the user the ability to manipulate fragments of a program's source as values. Their system performs type-checking to ensure that program compositions do not produce type errors, and verifies that phases of elaboration do not interfere with each other in ill-defined ways. Abstract structure operators [22], described in Section 2.3, were implemented using Compile-time Reflective ML (CRML) [23], a predecessor of MetaML.

Cardelli, et al. [6] describe a system that is similar to modern macro systems but admits more flexible syntaxes. Their approach is based on extensible grammars and does not incorporate a direct equivalent of our ellipses. They impose a type discipline on terms, which they use to ensure that elaboration terminates. Though their system allows users to extend the grammar of a language, the extensions are not defined in modular blocks (such as McMicMac's vocabularies) that can be combined with different base languages.

5 Conclusion and Future Work

We have shown how modern macros can be used to extend programming languages with constructs for specification mechanisms. Modern macro systems offer many features that make such extensions both convenient and powerful. First, macros support a simple and powerful shape-matching notation to define specifications. Second, macros automatically prevent insidious arrangements of code from affecting the lexical properties of both specification and user code. Third, macros can be grouped into modular constructs that make the macros more customizable and reusable. Most importantly, macros can track source locations and elaboration history and thus present users with helpful feedback, e.g., types, data flow and other messages, in terms of their source. This last issue, which is crucial for software development, is frequently ignored in the literature on software tools.

We anticipate many future directions for this work. First, our framework makes it easy to add program-processing tools. We can build tools that analyze, verify and document programs by selectively processing specifications that they understand, and falling back on the standard expansions for unfamiliar ones. Second, we expect the framework can accommodate other paradigms like adaptive programming [20]. One vocabulary could collect the adaptive specifications and program structure, while a second one would rewrite the program, removing the specifications and inserting code to perform traversals. Finally, by implement-

ing numerous platforms in this framework, we can study the properties that are common to structural specifications.

Acknowledgments We thank the anonymous referees for their detailed remarks.

References

1. Appel, A. W., B. F. Duba, D. B. MacQueen and A. P. Tolmach. Profiling in the presence of optimization and garbage collection. Technical Report CS-TR-197-88, Princeton University, November 1988.
2. Batory, D. and B. J. Geraci. Composition validation and subjectivity in GenVoca generators. *IEEE Transactions on Software Engineering*, pages 67–82, Feburary 1997.
3. Beck, K. and W. Cunningham. A laboratory for teaching object-oriented thinking. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, pages 1–6, 1989.
4. Bosch, J. Design patterns as language constructs. *Journal of Object-Oriented Programming*, 1998.
5. Budinsky, F. J., M. A. Finnie, J. M. Vlissides and P. S. Yu. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2), 1996.
6. Cardelli, L., F. Matthes and M. Abadi. Extensible syntax with lexical scoping. Research Report 121, Digital SRC, 1994.
7. Dybvig, R. K., R. Hieb and C. Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, December 1993.
8. Findler, R. B., C. Flanagan, M. Flatt, S. Krishnamurthi and M. Felleisen. DrScheme: A pedagogic programming environment for Scheme. In *Ninth International Symposium on Programming Languages, Implementations, Logics, and Programs*, pages 369–388, 1997.
9. Flatt, M. PLT MzScheme: Language manual. Technical Report TR97-280, Rice University, 1997.
10. Flatt, M., S. Krishnamurthi and M. Felleisen. Classes and mixins. In *Symposium on Principles of Programming Languages*, pages 171–183, January 1998.
11. Florijn, G., M. Meijers and P. van Winsen. Tool support for object-oriented patterns. In *European Conference on Object-Oriented Programming*, pages 472–495, 1997.
12. Gamma, E., R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Personal Computing Series. Addison-Wesley, Reading, MA, 1995.
13. Hedin, G. Language support for design patterns using attribute extension. In Bosch, J. and S. Mitchell, editors, *European Conference on Object-Oriented Programming Workshop Reader*, pages 137–140, 1997.
14. Helm, R., I. M. Holland and D. Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, pages 169–180, 1990.
15. Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, June 1997.

16. Kohlbecker, E. E., D. P. Friedman, M. Felleisen and B. F. Duba. Hygienic macro expansion. In *ACM Symposium on Lisp and Functional Programming*, pages 151–161, 1986.
17. Kohlbecker, E. E. and M. Wand. Macros-by-example: Deriving syntactic transformations from their specifications. In *Symposium on Principles of Programming Languages*, pages 77–84, 1987.
18. Krishnamurthi, S. PLT McMicMac: Elaborator manual. Technical Report 99-334, Rice University, Houston, TX, USA, 1999.
19. Lauder, A. and S. Kent. Precise visual specification of design patterns. In *European Conference on Object-Oriented Programming*, pages 114–134, July 1998.
20. Lieberherr, K. J. *Adaptive Object-Oriented Programming.* PWS Publishing, Boston, MA, USA, 1996.
21. Meijler, T. D., S. Demeyer and R. Engel. Making design patterns explicit in FACE, a framework for adaptive composition environment. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 94–110, September 1997.
22. Sheard, T. Automatic generation and use of abstract structure operators. *ACM Transactions on Programming Languages and Systems*, 13(4):531–557, October 1991.
23. Sheard, T. Guide to using CRML: Compile-time Reflective ML. Technical report, Department of Computer Science and Engineering, Oregon Graduate Institute, November 1992.
24. Smaragdakis, Y. and D. Batory. Implementing layered designs and mixin layers. In *European Conference on Object-Oriented Programming*, pages 550–570, July 1998.
25. Soukup, J. Implementing patterns. In *Pattern Languages of Program Design*, pages 395–412, 1995.
26. Taha, W. and T. Sheard. Multi-stage programming with explicit annotations. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 203–217, 1997.
27. Xerox PARC AspectJ Team. The AspectJ primer. Web document `http://www.parc.xerox.com/spl/projects/aop/aspectj/`, January 1999.

Polytypic Compact Printing and Parsing

Patrik Jansson[1] and Johan Jeuring[2]

[1] Computing Science, Chalmers University of Technology, Sweden
http://www.cs.chalmers.se/~patrikj/
patrikj@cs.chalmers.se
[2] Computer Science, Utrecht University, the Netherlands
http://www.cs.uu.nl/~johanj/
johanj@cs.uu.nl

Abstract. A generic compact printer and a corresponding parser are constructed. These programs transform values of any regular datatype to and from a bit stream. The algorithms are constructed along with a proof that printing followed by parsing is the identity. Since the binary representation is very compact, the printer can be used for compressing data - possibly supplemented with some standard algorithm for compressing bit streams. The compact printer and the parser are described in the polytypic Haskell extension PolyP.

1 Introduction

Many programs convert data from one format to another; examples are parsers, pretty printers, data compressors, encryptors, functions that communicate with a database, etc. Some of these programs, such as parsers and pretty printers, critically depend on the structure of the input data. Other programs, such as most data compressors and encryptors, more or less ignore the structure of the data. We claim that using the structure of the input data in a program for a data conversion problem almost always gives a more efficient program with better results. For example, a data compressor that uses the structure of the input data runs faster and compresses better than a conventional data compressor. This paper constructs (part of) a data compression program that uses the structure of the input data.

A lot of files that are distributed around the world, either over the internet or on CD-rom, possess structure — examples are databases, html files, and JavaScript programs — and it pays to compress these structured files to obtain faster transmission or fewer CD's. Structure-specific compression methods give much better compression results than conventional compression methods such as the Unix compress utility [3,17]. For example, Unix compress typically requires four bits per byte of Pascal program code, whereas Cameron [6] reports compression results of one bit per byte of Pascal program code. Algorithmic Research B.V. [5] sells compressors for structured data, and reports impressive results. Structured compression is also used in heap compression and binary I/O [16].

The basic idea of the structure-specific compression methods is simple: parse the input file into a structured value (an abstract syntax tree), and construct a

S.D. Swierstra (Ed.): ESOP/ETAPS'99, LNCS 1576, pp. 273–287, 1999.
© Springer-Verlag Berlin Heidelberg 1999

compact representation of the abstract syntax tree. For example, consider the datatype of binary trees

$$data\ Tree\ a = Leaf\ a \ \mid \ Bin\ (Tree\ a)\ (Tree\ a)$$

The following (rather artificial) example binary tree

$$\begin{aligned} &tree :: Tree\ () \\ &tree = Bin\ (Bin\ (Leaf\ ())\ (Bin\ (Leaf\ ())\ (Leaf\ ())))\ (Leaf\ ()) \end{aligned}$$

can be pretty printed to an (admittedly rather wasteful) text description of *tree* requiring 55 bytes. But since the datatype *Tree a* has two constructors, each constructor can be represented by a single bit. Furthermore, the datatype () has only one constructor so the single element can be represented by 0 bits. Thus we get the following representations:

```
Bin (Bin (Leaf ()) (Bin (Leaf ()) (Leaf ()))) (Leaf ())
 1    1      0          1      0          0           0
```

The compact representation consists of 7 bits, so only 1 byte is needed to store this tree. Of course, we are not always this lucky, but the average case is still very compact.

This idea has been around since the beginning of the 1980s, but as far as we are aware, there does not exist a general description of the program, only example instantiations appear in the literature. One of the goals of this paper is to describe the compact printing part, together with its inverse, of the compression program generically. It defines a *polytypic* program (a program that works for large classes of datatypes) for compact printing. Together with a parser generator this program is a generic description of the structured compression program. The implementation (as PolyP code) can be obtained from

`http://www.cs.chalmers.se/~patrikj/poly/`

The compression achieved by our compact printing algorithm, is through a compact representation of the structure of the data using only static information — the type of the data. Traditional (bit stream) compressors using dynamic (statistical) properties of the data are largely orthogonal to our approach and thus the best results are obtained by composing the compact printer with a bit stream compressor.

The fundamental property of the compact printing function *print* is that it has a left inverse[1]: the parsing function *parse*. This is a very common specification pattern: all of the example data conversion problems above are specified as pairs of inverse functions with some additional properties. Another example can be found in Haskell's prelude, which contains functions *show* and its inverse *read* of type:

$$\begin{aligned} show &:: Show\ a \ \Rightarrow \ a \rightarrow String \\ read &:: Read\ a \ \Rightarrow \ String \rightarrow a \end{aligned}$$

[1] That is, $parse \circ print = id$, but $print \circ parse$ need not be id. In the rest of the paper we will write just inverse, when we really mean left inverse.

Unfortunately, it is very hard to see from their definitions why *read* is the inverse of *show*. In this paper, the driving force behind the construction of the functions *print* and *parse* is inverse function construction. Thus correctness of *print* and *parse* is guaranteed by construction. Interestingly, when we forced ourselves to only construct pairs of inverse functions, we managed to reduce the size and complexity of the resulting program considerably compared with our previous attempts.

A second desired property of the compact printing function is that given an element x, the length of *print* x is less than the length of *prettyprint* x, where *prettyprint* is a function that prints a value in a standard fashion, like the *show* function of Haskell. This is in general difficult or impossible to prove, and beyond the scope of this paper. More information can be found in the literature [15].

Summarising, this paper has the following goals:

- construct a polytypic compact printing program together with its inverse;
- show how to construct and calculate with polytypic functions;
- take a first step towards a theory of polytypic data conversion.

This paper is organised as follows. Section 2 briefly introduces polytypic programming. Section 3 defines some basic types and classes, and introduces the compact printing program. Section 4 sketches the construction and correctness proof of the compact printing program. Section 5 concludes. Appendix A describes the laws we need in the proofs.

2 Polytypic programming

The compact printing and parsing functions are polytypic functions. This section briefly introduces polytypic functions in the context of the Haskell extension PolyP [9], and defines some basic polytypic concepts used in the paper. We assume that the reader is familiar with the initial algebra approach to datatypes, and not completely unfamiliar with polytypic programming. For an introduction to polytypic programming, see [1,10].

A polytypic function is a function parametrised on type constructors. Polytypic functions are defined either by induction on the structure of user-defined datatypes, or defined in terms of other polytypic (and non-polytypic) functions. In the definition of a function that works for an arbitrary (as yet unknown) datatype we cannot use the constructors to build values, nor to pattern match against values. Instead, we use two built-in functions, *inn* and *out*, to construct and destruct a value of an arbitrary datatype from and to its top level components. With a recursive datatype $d\,a$ as a fixed point of a pattern functor $\Phi_d\,a$, *inn* and *out* are the fold and unfold isomorphisms showing $d\,a \cong \Phi_d\,a\,(d\,a)$.

$$\begin{aligned} &\mathit{inn} :: \mathit{Regular}\ d \;\Rightarrow\; (\mathit{FunctorOf}\ d)\ a\ (d\ a) \to d\ a \\ &\mathit{out} :: \mathit{Regular}\ d \;\Rightarrow\; d\ a \to (\mathit{FunctorOf}\ d)\ a\ (d\ a) \end{aligned}$$

The pattern functor is used to capture the (top level) structure of a datatype, for example, a list is either empty or contains one element and a recursive occurrence

of a list. Hence: $FunctorOf\ List = Empty + (Par * Rec)$. Similarly, the pattern functor of the datatype $Tree\ a$ is $Par + (Rec * Rec)$. As a last example, the datatype $Rose\ a$ of rose trees over a:

$$data\ Rose\ a = Node\ a\ (List\ (Rose\ a))$$

has the pattern functor $FunctorOf\ Rose = Par * (List\ @\ Rec)$, where @ denotes functor composition. In general, PolyP's pattern functors are generated by the following grammar:

$$f, g, h \ ::= \ g + h \ \mid \ g * h \ \mid \ Empty \ \mid \ Par \ \mid \ Rec \ \mid \ d\ @\ g \ \mid \ Const\ t$$

where d generates regular datatype constructors, and t generates types. The pattern functor $Const\ t$ denotes a constant functor with value t. The type context $Bifunctor\ f \ \Rightarrow$ is used to indicate that f is a pattern functor.

Using the **polytypic** construct a polytypic function can be defined by induction over the structure of pattern functors. As an example we take the function *psum* defined in figure 1. (The subscripts indicating the type are included for

```
psum :: Regular d ⇒ d Int → Int
psum = fsum ∘ fmap id psum ∘ out

polytypic fsum_f :: Bifunctor f ⇒ f Int Int → Int
  = case f of
      g + h   ⟶ either fsum_g fsum_h
      g * h   ⟶ λ(x, y) → fsum_g x + fsum_h y
      Empty   ⟶ λ() → 0
      Par     ⟶ λn → n
      Rec     ⟶ λs → s
      d @ g   ⟶ psum_d ∘ (pmap_d fsum_g)
      Const t ⟶ λx → 0

pmap  :: Regular d ⇒ (a → b) → d a → d b
fmap  :: Bifunctor f ⇒ (a → c) → (b → d) → f a b → f c d
```

Fig. 1. The definition of *psum*

readability and are not part of the definition.) Function *psum* sums the integers in a datatype with integers. We use the naming convention that recursive polytypic functions start with a '*p*' (as in polytypic) but non-recursive polytypic definitions start with an '*f*' (as in functor). The 'polytypic map', *pmap*, takes a function $f :: a \to b$ and a value $x :: d\ a$, and applies f to all a values in x, giving a value of type $d\ b$. The 'functor map', *fmap*, takes two functions $g :: a \to c$ and $h :: b \to d$ and a value $x :: f\ a\ b$, and applies g to all a values in x, and h to all b values in x, giving a value of type $f\ c\ d$. The definitions of *pmap* and *fmap* can be found in the distribution of PolyP.

Note that function *psum* is only defined for *Regular* datatypes $d\ a$. A datatype $d\ a$ is regular (satisfies *Regular* d) if it contains no function spaces, and if the argument of the type constructor d is the same on the left- and right-hand side of its definition. In the rest of the paper we always assume that $d\ a$ is a regular datatype and that f is a pattern functor but we omit the contexts (*Regular* $d\ \Rightarrow$ or *Bifunctor* $f\ \Rightarrow$) from the types for brevity.

3 Basic types and classes

Compact printing. A natural choice for the type of a compact printing function for type a is $a \to \mathit{Text}$, where *Text* is the type of printed values, for example *String* or [*Bit*]. Since we want to define *print* as a recursive function, this would lead to quadratic behaviour when repeatedly concatenating intermediate results. The standard solution for printing functions is to add an accumulating parameter (to which the output is prepended) thus changing the type to $a \to \mathit{Text} \to \mathit{Text}$, or equivalently, to $(a, \mathit{Text}) \to \mathit{Text}$.

Parsing. Parsing is the inverse of printing, and hence a first approximation of its type is $\mathit{Text} \to a$. Since we want to apply parsers one after the other, we need both a parsed result and the remaining part of the input string, which can be passed to the next parser. The standard solution for parsing functions is to change the type to $\mathit{Text} \to (a, \mathit{Text})$.

Side effects as functions. We can make the types for printing and parsing more symmetric by pairing the single *Text* component with a unit type to get the isomorphic type $(a, \mathit{Text}) \to ((), \mathit{Text})$ for printing and $((), \mathit{Text}) \to (a, \mathit{Text})$ for parsing. Both these types are instances of the more general type *TextStateArr* $a\ b$:

$$\textbf{newtype}\ \mathit{TextStateArr}\ a\ b = \mathit{TS}\ ((a, \mathit{Text}) \to (b, \mathit{Text}))$$

An element of type *TextStateArr* $a\ b$ models a function that takes a value of type a and returns a value of type b, and possibly has a side effect on the state *Text*. Thus a compact printer (for a-values) has type *TextStateArr* $a\ ()$, and a corresponding parser has type *TextStateArr* $()\ a$.

The first steps. Our goal is to construct two functions and a proof:

– A function *pc* ('polytypic compacting') that takes a compact printing program on the element level a to a compact printing program on the datatype level $d\ a$:

$$\mathit{pc} :: \mathit{TextStateArr}\ a\ () \to \mathit{TextStateArr}\ (d\ a)\ ()$$

For example, the function that compresses the tree in the introductory section is obtained by instantiating the polytypic function *pc* to *Tree* and applying the instance to a (trivial) compact printing program for the type ().

– A function *pu* ('polytypic uncompacting') that takes a parsing program on the element level a to a parsing program on the datatype level $d\ a$:

$$pu :: TextStateArr\ ()\ a \to TextStateArr\ ()\ (d\ a)$$

  For the *Tree* example the element level parsing program is a function that parses nothing, and returns (), the value of type ().
– A proof that if c and u are inverses on the element level a, $pc\ c$ and $pu\ u$ are inverses on the datatype level $d\ a$.

In the following section, instead of using the type $TextStateArr\ a\ b$ in the definitions of *pc* and *pu*, we will use the more abstract type $a \rightsquigarrow b$, where $(\rightsquigarrow)$ is an *arrow* type constructor.

The class *Arrow*. The type $TextStateArr\ a\ b$ encapsulates functions from a to b that manipulate a state of type *Text*. Since a parser could easily use a more complicated type, for example to store statically available information [14], and also the printer could use a more complicated type, we will go one step further in the abstraction by introducing the constructor class *Arrow* [8]:

$$\begin{aligned}
&\textbf{class}\ Arrow\ (\rightsquigarrow)\ \textbf{where}\\
&\quad arr\ \ :: (a \to b) \to (a \rightsquigarrow b)\\
&\quad (\ggg)\ :: (a \rightsquigarrow b) \to (b \rightsquigarrow c) \to (a \rightsquigarrow c)\\
&\quad (\lessdot)\ :: (a \rightsquigarrow c) \to (b \rightsquigarrow d) \to (Either\ a\ b \rightsquigarrow Either\ c\ d)\\
&\quad first\ :: (a \rightsquigarrow b) \to ((a, c) \rightsquigarrow (b, c))
\end{aligned}$$

The method *arr* of the class *Arrow* embeds functions as arrows and $arr\ id$ together with $(\ggg)$ form the signature of the category with types as objects, and elements of $a \rightsquigarrow b$ as arrows from a to b. This category has a binary (sum) functor (the method $(\lessdot)$) and a "half-product" functor (*first*). Below we write $\overrightarrow{f}$ as a shorthand notation for $arr\ f$. In the appendix we formalise the properties we need from *Arrow*s to construct the definitions of functions *pc* and *pu* along with the proof of their correctness.

As an example of programming with arrows, we define *second* — the other half-product — in terms of *first* :

$$\begin{aligned}
&second\ :: (a \rightsquigarrow b) \to ((c, a) \rightsquigarrow (c, b))\\
&second\ f = \overrightarrow{swap} \ggg first\ f \ggg \overrightarrow{swap}\\
&\\
&swap :: (a, b) \to (b, a)\\
&swap\ (a, b) = (b, a)
\end{aligned}$$

Using *first* and *second* we can define two candidates for being product functors, but when the arrows have side-effects, neither of these are functors as they fail to preserve composition.

$$\begin{aligned}
(\ast\!\!\ast\!\!\ast_i) &:: (a \rightsquigarrow c) \to (b \rightsquigarrow d) \to ((a, b) \rightsquigarrow (c, d))\\
f \ast\!\!\ast\!\!\ast_1 g &= first\ f \ggg second\ g\\
f \ast\!\!\ast\!\!\ast_2 g &= second\ f \ggg first\ g
\end{aligned}$$

Type constructor *TextStateArr* can be made an instance of *Arrow* as follows:

$$
\begin{array}{l}
\mathit{mapFst} :: (a \to b) \to (a, c) \to (b, c) \\
\mathit{mapFst}\, f\, (a, c) = (f\ a, c)
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Arrow}\ \mathit{TextStateArr}\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{arr}\, f & = & \mathit{TS}\, (\mathit{mapFst}\, f) \\
\mathit{TS}\, f \ggg \mathit{TS}\, g & = & \mathit{TS}(g \circ f) \\
\mathit{TS}\, f \lhd\rhd \mathit{TS}\, g & = & \mathit{TS}\, (\lambda(x, t) \to \mathit{either}\ (\lambda a \to \mathit{mapFst}\ \mathit{Left}\, (f\, (a, t))) \\
 & & \qquad\qquad (\lambda b \to \mathit{mapFst}\ \mathit{Right}\, (g\, (b, t))) \\
 & & \qquad\qquad x) \\
\mathit{first}\, (\mathit{TS}\, f) & = & \mathit{TS}\, (\lambda((a, c), t) \to \textbf{let}\ (b, t') = f\, (a, t) \\
 & & \qquad\qquad\qquad\ \textbf{in}\ \ ((b, c), t'))
\end{array}
\end{array}
$$

Printing constructors. To construct the printer and the parser we need a little more structure than provided by the *Arrow* class – we need a way of handling constructors. Since a constructor can be coded by a single natural number, we can use a class *ArrowNat* to characterise arrows that have operations for printing and parsing constructor numbers:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Arrow}\, (\rightsquigarrow)\ \Rightarrow\ \mathit{ArrowNat}\, (\rightsquigarrow)\ \textbf{where} \\
\quad \mathit{printCon} :: \mathit{Nat} \rightsquigarrow () \\
\quad \mathit{parseCon} :: () \rightsquigarrow \mathit{Nat} \\
\quad \texttt{--}\ \text{Requirement: } \mathit{printCon} \ggg \mathit{parseCon} = \overrightarrow{\mathit{id}}
\end{array}
$$

With $\mathit{Text} = [\mathit{Nat}]$, the instances for *TextStateArr* are straightforward, and the printing algorithm constructed in the following section will in its simplest form just output a list of numbers given an argument tree of any type. A better solution is to code these numbers as bits and here we have some choices on how to proceed. We could decide on a fixed maximal size for numbers and store them using their binary representation but, as most datatypes have few constructors, this would waste space. We will instead statically determine the number of constructors in the datatype and code every single number in only as many bits as needed. For an n-constructor datatype we use just $\lceil \log_2 n \rceil$ bits to code a constructor. An interesting effect of this coding is that the constructor of any single constructor datatype will be coded using 0 bits! We obtain better results if we use Huffman coding with equal probabilities for the constructors, resulting in a variable number of bits per constructor. Even better results are obtained if we analyse the datatype, and give different probabilities to the different constructors. However, our goal is not to squeeze the last bit out of our data, but rather to show how to construct the polytypic program. Since the number of bits used per constructor depends on the type of the value that is compressed, *printCon* and *parseCon* need in general be polytypic functions. Their definitions are omitted, but can be found in the code on the web page for this paper.

In the sequel $(\rightsquigarrow)$ will always stand for an arrow type constructor in the class *ArrowNat* but, as with *Regular*, we often omit the type context for brevity.

4 The construction of the program

We want to construct a function *pc* that takes a compact printing program on the element level a to a compact printing program on the datatype level $d\ a$, together with a parsing function *pu*, which takes a compact parsing program on the element level a to a compact parsing program on the datatype level $d\ a$, and a proof that *pu* is the inverse of *pc*:

$$pc :: (a \rightsquigarrow ()) \to (d\ a \rightsquigarrow ())$$
$$pu :: (() \rightsquigarrow a) \to (() \rightsquigarrow d\ a)$$

$$c \ggg u = id \quad \Rightarrow \quad pc\ c \ggg pu\ u = pid\ (c \ggg u) = id \tag{1}$$

In the proofs below we will assume that the arrows c and u satisfy $c \ggg u = id$.

Overview of the construction. The construction can be interpreted either as fusing the printer $pc\ c$ with the parser $pu\ u$ to get an identity arrow *id* or, equivalently, as splitting the identity arrow into a composition of a printer and a parser. As both the printer and the parser are polytypic functions, and both lift an argument level arrow to a datatype level arrow, we start by presenting a polytypic "identity function" *pid* that lifts an element level identity arrow to a datatype level identity arrow. Function *pid* is constructed below together with *pc* and *pu* and the proof of equation 1 but the resulting definition is presented already here, in figure 2, to aid the reading. The proof that $pid\ id = id$ is simple

```
pid  :: (a ⇝ b) → (d a ⇝ d b)
pid i = out⃗ ⋙ fid i (pid i) ⋙ inn⃗

polytypic fid :: (a ⇝ c) → (b ⇝ d) → (f a b ⇝ f c d)
  = λi j → case f of
             g + h   ⟶  fid i j ⍏ fid i j
             g * h   ⟶  (fid i j) ⁂₁ (fid i j)
             Empty   ⟶  id⃗
             Par     ⟶  i
             Rec     ⟶  j
             d @ g   ⟶  pid (fid i j)
```

Fig. 2. The definition of *pid* and *fid*.

and omitted. As we are defining polytypic functions the construction follows the structure of regular datatypes: A regular datatype is a fix-point of a pattern functor, the pattern functor is a sum of products, and the products can involve type parameters, other types, etc.

The arrow $pc\ c$ prints a compact representation of a value of type $d\ a$. It does this by recursing over the value, printing each constructor by computing

its constructor number, and each element by using the argument printer c. The constructor number is computed by means of function *fcSum*, which also takes care of passing on the recursion to the children. An arrow *printCon* prints the constructor number with the correct number of bits. Finally, function *fcProd* makes sure the information is correctly threaded through the children.

Top level recursion. We want function *pc* to be 'on-line' or lazy: it should output compactly printed data immediately, and given part of the compactly printed data, *pu* should reconstruct part of the input value. Thus functions *pc* and *pu* can also be used to compactly print infinite streams, for example. We have not been able to define function *pc* with a standard recursion operator such as the catamorphism: threading the side effects in the right order turned out to be a problem. Instead of a recursion operator we use explicit recursion on the top level, guided by *fc* and *fu*.

As *pc* decomposes its input value, and compactly prints the constructor and the children by means of a function *fc* (defined below), *pu* must do the opposite: first parse the components using *fu* and then construct the top level value:

$$\begin{aligned} pc\ c &= fc\ c\ (pc\ c) \lll \overrightarrow{out} \\ pu\ u &= fu\ u\ (pu\ u) \ggg \overrightarrow{inn} \end{aligned}$$

Here $f \lll g \stackrel{def}{=} g \ggg f$ is used to reveal the symmetry of the definitions. Thus we need two new functions, *fc* and *fu*, and we can already guess that we will need a corresponding fusion law:

$$\begin{aligned} fc &:: (a \rightsquigarrow ()) \to (b \rightsquigarrow ()) \to (f\ a\ b \rightsquigarrow ()) \\ fu &:: (() \rightsquigarrow a) \to (() \rightsquigarrow b) \to (() \rightsquigarrow f\ a\ b) \end{aligned}$$

$$fc\ c\ c' \ggg fu\ u\ u' = fid\ (c \ggg u)\ (c' \ggg u') \tag{2}$$

We will use the following variant of fixed-point fusion [12,13]:[2]

$$\mu f \ggg \mu g = \mu h \quad \Leftarrow \quad f\ c' \ggg g\ u' = h\ (c' \ggg u') \tag{3}$$

Given (2) we can now prove (1).

$$\begin{array}{ll} & pc\ c \ggg pu\ u = pid\ (c \ggg u) \\ \Leftarrow & \text{Definitions of } pc,\ pu, \text{ fixed-point theorem (3)} \\ & \overrightarrow{out} \ggg fc\ c\ c' \ggg fu\ u\ u' \ggg \overrightarrow{inn} = h\ (c' \ggg u') \\ \equiv & \text{Equation (2).} \\ & \overrightarrow{out} \ggg fid\ (c \ggg u)\ (c' \ggg u') \ggg \overrightarrow{inn} = h\ (c' \ggg u') \\ \equiv & \bullet \text{ Define } h\ j = \overrightarrow{out} \ggg fid\ (c \ggg u)\ j \ggg \overrightarrow{inn} \\ & True \end{array}$$

The resulting definition of function *pid* can be found in figure 2.

[2] Strictly speaking the variables c' and u' on the right hand side of the implication should be $\forall$-quantified over $\{f^i \perp \mid i \in \mathbf{N}\}$ and $\{g^i \perp \mid i \in \mathbf{N}\}$ respectively.

Printing constructors. We want to construct functions *fc* and *fu* such that (2) holds. Furthermore, these functions should do the actual compact printing and parsing of the constructors using $printCon :: Nat \rightsquigarrow ()$ and $parseCon :: () \rightsquigarrow Nat$ from the *ArrowNat* class:

$$fc\ c\ c' = printCon \lll fcSum\ c\ c'$$
$$fu\ u\ u' = parseCon \ggg fuSum\ u\ u'$$

The arrow $fcSum\ c\ c'$ prints a value (using the argument printers c and c' for the parameters and the recursive structures, respectively) and returns the number of the top level constructor, by determining the position of the constructor in the pattern functor (a sum of products). The arrow *printCon* prepends the constructor number to the output. As $printCon \ggg parseCon = \overrightarrow{id}$ by assumption, the requirement that function *fu* can be fused with *fc* is now passed on to *fuSum* and *fcSum*:

$$fcSum :: (a \rightsquigarrow ()) \rightarrow (b \rightsquigarrow ()) \rightarrow (f\ a\ b \rightsquigarrow Nat)$$
$$fuSum :: (() \rightsquigarrow a) \rightarrow (() \rightsquigarrow b) \rightarrow (Nat \rightsquigarrow f\ a\ b)$$

$$fcSum\ c\ c' \ggg fuSum\ u\ u' = fid\ (c \ggg u)\ (c' \ggg u') \tag{4}$$

The arrow *parseCon* reads the constructor number and passes it on to the arrow $fuSum\ u\ u'$ which selects the desired constructor and uses its argument parsers u and u' to fill in the parameter and recursive component slots in the functor value.

Calculating constructor numbers. The pattern functor of a Haskell datatype with n constructors is an n-ary sum (of products) on the outermost level. This sum is in PolyP represented by a nested binary sum, which associates to the right. Consequently, we define *fcSum* by induction over the nested sum part of the pattern functor and defer the handling of the product part to *fcProd*:

$$\begin{aligned}
&\textbf{polytypic}\ fcSum :: (a \rightsquigarrow ()) \rightarrow (b \rightsquigarrow ()) \rightarrow (f\ a\ b \rightsquigarrow Nat)\\
&\quad = \lambda c\ c' \rightarrow \textbf{case}\ f\ \textbf{of}\\
&\qquad\qquad g + h \longrightarrow (fcProd\ c\ c' \lhd\!\rhd fcSum\ c\ c') \ggg \overrightarrow{inn_{Nat}}\\
&\qquad\qquad\ \ g \longrightarrow fcProd\ c\ c' \ggg \lambda() \rightarrow 0
\end{aligned}$$

$$\begin{aligned}
&\textbf{polytypic}\ fuSum :: (() \rightsquigarrow a) \rightarrow (() \rightsquigarrow b) \rightarrow (Nat \rightsquigarrow f\ a\ b)\\
&\quad = \lambda u\ u' \rightarrow \textbf{case}\ f\ \textbf{of}\\
&\qquad\qquad g + h \longrightarrow (fuProd\ u\ u' \lhd\!\rhd fuSum\ u\ u') \lll \overrightarrow{out_{Nat}}\\
&\qquad\qquad\ \ g \longrightarrow fuProd\ u\ u' \lll \lambda 0 \rightarrow ()
\end{aligned}$$

The types for *fcProd* and *fuProd* and the corresponding fusion law are unsurprising:

$$fcProd :: (a \rightsquigarrow ()) \rightarrow (b \rightsquigarrow ()) \rightarrow (f\ a\ b \rightsquigarrow ())$$
$$fuProd :: (() \rightsquigarrow a) \rightarrow (() \rightsquigarrow b) \rightarrow (() \rightsquigarrow f\ a\ b)$$

$$fcProd\ c\ c' \ggg fuProd\ u\ u' = fid\,(c \ggg u)\,(c' \ggg u') \tag{5}$$

We prove equation (4) by induction over the nested sum structure of the functor. The induction hypothesis is that (4) holds for the $fcSum_h$.

The sum case: $g + h$

$$\begin{aligned}
& fcSum_{g+h}\ c\ c' \ggg fuSum_{g+h}\ u\ u' \\
= \quad & \text{Definitions} \\
& (fcProd_g\ c\ c' \mathbin{<\!\!+\!\!>} fcSum_h\ c\ c') \ggg \overrightarrow{inn_{Nat}} \ggg \\
& \overrightarrow{out_{Nat}} \ggg (fuProd_g\ u\ u' \mathbin{<\!\!+\!\!>} fuSum_h\ u\ u') \\
= \quad & out_{Nat} \circ inn_{Nat} = id \\
& (fcProd_g\ c\ c' \mathbin{<\!\!+\!\!>} fcSum_h\ c\ c') \ggg (fuProd_g\ u\ u' \mathbin{<\!\!+\!\!>} fuSum_h\ u\ u') \\
= \quad & (\mathbin{<\!\!+\!\!>}) \text{ is a bifunctor} \\
& (fcProd_g\ c\ c' \ggg fuProd_g\ u\ u') \mathbin{<\!\!+\!\!>} (fcSum_h\ c\ c' \ggg fuSum_h\ u\ u') \\
= \quad & \text{Equation (5) and the induction hypothesis} \\
& fid_g\,(c \ggg u)\,(c' \ggg u') \mathbin{<\!\!+\!\!>} fid_h\,(c \ggg u)\,(c' \ggg u') \\
= \quad & \bullet \text{ Define } fid_{g+h} \\
& fid_{g+h}\,(c \ggg u)\,(c' \ggg u')
\end{aligned}$$

The base case: g

$$\begin{aligned}
& fcProd_g\ c\ c' \ggg \lambda() \to 0 \ggg \lambda 0 \to () \ggg fuProd_g\ u\ u' \\
= \quad & \overrightarrow{\lambda() \to 0} \ggg \overrightarrow{\lambda 0 \to ()} = id :: () \rightsquigarrow () \\
& fcProd_g\ c\ c' \ggg fuProd_g\ u\ u' \\
= \quad & \text{Equation (5)} \\
& fid_g\,(c \ggg u)\,(c' \ggg u')
\end{aligned}$$

Sequencing the parameters. The last part of the construction of the program is the two functions *fcProd* and *fuProd* defined in figure 3. The earlier functions have calculated and printed the constructors, so what is left is "arrow plumbing". The arrow $fcProd\ c\ c'$ traverses the top level structure of the data and inserts the correct compact printers: c at argument positions and c' at substructure positions. The structure of *fuProd* is very similar but as it is the inverse of *fcProd*, all arrows are composed in the opposite order. The inverse proof is a relatively straightforward induction over the pattern functor structure, but omitted here due to space constraints.

```
polytypic fcProd :: (a ⇝ ()) → (b ⇝ ()) → (f a b ⇝ ())
  = λc c' → case f of
              g ∗ h   ⟶ (fcProd c c') ⁂₂ (fcProd c c') ⋙ λ((), ()) → ()
              Empty   ⟶ id
              Par     ⟶ c
              Rec     ⟶ c'
              d @ g   ⟶ pc (fcProd c c')

polytypic fuProd :: (() ⇝ a) → (() ⇝ b) → (() ⇝ f a b)
  = λu u' → case f of
              g ∗ h   ⟶ (fuProd u u') ⁂₁ (fuProd u u') ⋘ λ() → ((), ())
              Empty   ⟶ id
              Par     ⟶ u
              Rec     ⟶ u'
              d @ g   ⟶ pu (fuProd u u')
```

Fig. 3. The definition of *fcProd* and *fuProd*.

5 Conclusions

Results

- We have constructed a polytypic program for compact printing and parsing of structured data. As far as we are aware, this is the first generic description of a program for compact printing (structured data compression).
- The pair of functions for compact printing and parsing are inverse functions by construction. Since we started applying the inverse function requirement rigorously in the construction of the program, the size and the complexity of the code have been reduced considerably. We think that such a rigorous approach is the only way to obtain elegant solutions to involved polytypic problems.
  Another concept that simplified the construction and form of the program is arrows. In our first attempts we used monads instead of arrows. Although it is perfectly well possible to construct the compact printing and parsing functions with monads [7], the inverse function construction, and hence the correctness proof, is much simpler with arrows.
- We have shown how to convert data to and from a bit stream. This is an example of a data conversion program, and we hope that the construction in this paper is reusable in solutions for other data conversion problems.

Future work

- The current program produces compact, but not human-readable, output. A pretty printer for structured data has a very similar structure, and we want to investigate how to introduce the right abstractions to obtain a single program for both pretty printing and compact printing of structured data.

– In the future we want to investigate whether or not relations can help to simplify the construction even more, by specifying compact printing as a relation, and letting parsing be its relational converse [2,4].
  We have presented a calculation of a polytypic program. We think that calculating with polytypic functions is still rather cumbersome, and we hope to obtain more theory, in the style of [11], to further simplify calculations with polytypic programs.
– We want to construct polytypic programs for other data conversion problems such as encryption and database communication.

Acknowledgements. Roland Backhouse helped with the fixed point calculation. Joost Halenbeek implemented a polytypic data compression program using monads. The anonymous referees suggested many improvements for contents and presentation.

References

1. R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming — an introduction. To appear in AFP'98, LNCS, Springer-Verlag, 1998.
2. R.C. Backhouse, P.J. de Bruin, P. Hoogendijk, G. Malcolm, T.S. Voermans, and J.C.S.P. van der Woude. Relational catamorphisms. In B. Möller, editor, *Constructing Programs from Specifications*, pages 287–318. North-Holland, 1991.
3. Timothy C. Bell, John G. Cleary, and Ian H. Witten. *Text Compression*. Prentice Hall, 1990.
4. R.S. Bird and O. de Moor. *Algebra of Programming*. Prentice-Hall International, 1996.
5. Algorithmic Research B.V. SDR compression products. See `http://www.algoresearch.com/`, 1998.
6. Robert D. Cameron. Source encoding using syntactic information source models. *IEEE Transactions on Information Theory*, 34(4):843–850, 1988.
7. J. Halenbeek. Comparing approaches to generic programming. Master's thesis, Department of Computer Science, Utrecht University, 1998. To appear.
8. John Hughes. Generalising monads. Invited talk at MPC'98, 1998. Slides available from `http://www.md.chalmers.se/Conf/MPC98/talks/JohnHughes/`.
9. P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *POPL'97*, pages 470–482. ACM Press, 1997.
10. J. Jeuring and P. Jansson. Polytypic programming. In *AFP'96*, volume 1129 of *LNCS*, pages 68–114. Springer-Verlag, 1996.
11. L. Meertens. Calculate polytypically! In *PLILP'96*, volume 1140 of *LNCS*, pages 1–16. Springer Verlag, 1996.
12. Erik Meijer. *Calculating compilers*. PhD thesis, Nijmegen University, 1992.
13. Mathematics of Program Construction Group (Eindhoven Technical University). Fixed-point calculus. *Information Processing Letters*, 53(3):131–136, 1995.
14. Swierstra S.D. and L. Duponcheel. Deterministic, error-correcting combinator parsers. In John Launchbury, Erik Meijer, and Tim Sheard, editors, *Advanced Functional Programming*, LNCS 1129, pages 184–207. Springer-Verlag, 1996.
15. R.G. Stone. On the choice of grammar and parser for the compact analytical encoding of programs. *The Computer Journal*, 29(4):307–314, 1986.

16. M. Wallace and C. Runciman. Heap compression and binary I/O in haskell. In *2nd ACM Haskell Workshop*, 1997.
17. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

A Properties of *Arrow*s

The properties we need from an *Arrow* type constructor for the definitions of the generic printer and parser are most succinctly described using category theoretic terminology. We work in a base category $\mathcal{C}$ of (Haskell-) types and functions and a type constructor $\rightsquigarrow$ is an *Arrow* if we have a category $\mathcal{A}$ with the same types as objects, but with elements of $a \rightsquigarrow b$ as arrows:

$$\mathcal{C} = (H, (\rightarrow), (\circ), id)$$
$$\mathcal{A} = (H, (\rightsquigarrow), (\lll), id)$$

Furthermore $\mathcal{A}$ must have a binary (sum) functor $((\lhd\rhd) : \mathcal{A} \times \mathcal{A} \rightarrow \mathcal{A})$, "half-product" functors $(first_c : \mathcal{A} \rightarrow \mathcal{A})$ and there must be a functor $(\overrightarrow{\cdot} : \mathcal{C} \rightarrow \mathcal{A})$ lifting functions to arrows. A set of laws sufficient for the proof of the correctness of the *print-parse*-pair is given in figure 4. We do not require the stronger

$\mathcal{A}$ is a category	$id \lll f = f = f \lll id$	$(\lll, id)$
	$(f \lll g) \lll h = f \lll (g \lll h)$	$(\lll, \lll)$
$\overrightarrow{\cdot} : \mathcal{C} \rightarrow \mathcal{A}$	$\overrightarrow{id} = id$	$(\overrightarrow{\cdot}, id)$
$\overrightarrow{a} = a$	$\overrightarrow{f \circ g} = \overrightarrow{f} \lll \overrightarrow{g}$	$(\overrightarrow{\cdot}, \lll)$
$(\lhd\rhd) : \mathcal{A} \times \mathcal{A} \rightarrow \mathcal{A}$	$id \lhd\rhd id = id$	$(\lhd\rhd, id)$
$a \lhd\rhd b = Either\ a\ b$	$(f \lhd\rhd g) \lll (f' \lhd\rhd g') = (f \lll f') \lhd\rhd (g \lll g')$	$(\lhd\rhd, \lll)$
$first_c : \mathcal{A} \rightarrow \mathcal{A}$	$first_c\ id = id$	$(first_c, id)$
$first_c\ a = (a, c)$	$first_c\ (f \lll g) = first_c\ f \lll first_c\ g$	$(first_c, \lll)$
	$first_c\ (f \lhd\rhd g) = first_c\ f \lhd\rhd first_c\ g$	$(first_c, \lhd\rhd)$

Fig. 4. Laws for *Arrow*s.

requirements that $(\lhd\rhd)$ should be a true categorical sum or that $first_c$ (combined with $second_c$) should give a categorical product as this would rule out many useful arrow type constructors. In fact, the proof goes through even with slightly weaker conditions on the arrows than those in figure 4, and thus we may be able to extend the class of possible arrows further.

We denote reverse composition in $\mathcal{A}$ with $(\ggg)$ and we often use the obvious variants of the laws for this operator. When translating the *Arrow* requirements back to a Haskell class we omit id as it is equal to $\overrightarrow{id}$. The resulting code is shown in figure 5 where we also introduce some useful abbreviations.

```
class Arrow a where
  arr    ::  (b -> c) -> a b c
  (>>>)  ::  a b c -> a c d -> a b d
  (|||)  ::  a b d -> a c d -> a (Either b c) d
  (<+>)  ::  a b d -> a c e -> a (Either b c) (Either d e)
  first  ::  a b c -> a (b,d) (c,d)
  second ::  a b c -> a (d,b) (d,c)
  -- Defaults:
  f <+> g = (f >>> arr Left) ||| (g >>> arr Right)
  f ||| g = (f <+> g) >>> arr (either id id)

  second f = arr swap >>> first f  >>> arr swap
  first  f = arr swap >>> second f >>> arr swap

-- Utilities
(<<<) :: Arrow a => a c d -> a b c -> a b d
g <<< f = f >>> g

swap :: (a,b) -> (b,a)
swap   ~(x,y) =  (y,x)

data Nat = Z | S Nat

innNat :: Either () Nat -> Nat
innNat  = either (const Z) S

outNat :: Nat   ->  Either () Nat
outNat    (Z)    =  Left   ()
outNat    (S n)  =  Right     n

-- Either and either are predefined in Haskell
data Either a b  =  Left a | Right b

either :: (a -> c) -> (b -> c) -> Either a b -> c
either f g (Left x)   =  f x
either f g (Right x)  =  g x
```

Fig. 5. The *Arrow* operations as Haskell code.

Dynamic Programming via Static Incrementalization *

Yanhong A. Liu and Scott D. Stoller

Computer Science Department, Indiana University, Bloomington, IN 47405
{liu,stoller}@cs.indiana.edu

Abstract. Dynamic programming is an important algorithm design technique. It is used for solving problems whose solutions involve recursively solving subproblems that share subsubproblems. While a straightforward recursive program solves common subsubproblems repeatedly and often takes exponential time, a dynamic programming algorithm solves every subsubproblem just once, saves the result, reuses it when the subsubproblem is encountered again, and takes polynomial time. This paper describes a systematic method for transforming programs written as straightforward recursions into programs that use dynamic programming. The method extends the original program to cache all possibly computed values, incrementalizes the extended program with respect to an input increment to use and maintain all cached results, prunes out cached results that are not used in the incremental computation, and uses the resulting incremental program to form an optimized new program. Incrementalization statically exploits semantics of both control structures and data structures and maintains as invariants equalities characterizing cached results. The principle underlying incrementalization is general for achieving drastic program speedups. Compared with previous methods that perform memoization or tabulation, the method based on incrementalization is more powerful and systematic. It has been implemented and applied to numerous problems and succeeded on all of them.

1 Introduction

Dynamic programming is an important technique for designing efficient algorithms [2,46,14]. It is used for problems whose solutions involve recursively solving subproblems that overlap. While a straightforward recursive program solves common subproblems repeatedly, a dynamic programming algorithm solves every subproblem just once, saves the result in a table, and reuses the result when the subproblem is encountered again. This can reduce the time complexity from exponential to polynomial. The technique is generally applicable to all problems whose efficient solutions involve memoizing results of subproblems [4,5].

Given a straightforward recursion, there are two traditional ways to achieve the effect of dynamic programming [14]: memoization [34] and tabulation [5].

Memoization uses a mechanism that is separate from the original program to save the result of each function call or reduction [34,19,22,35,24,43,45,39,25,18,1].

* This work is supported in part by NSF under Grant CCR-9711253 and ONR under Grant N0014-99-1-0132.

S.D. Swierstra (Ed.): ESOP/ETAPS'99, LNCS 1576, pp. 288–305, 1999.
© Springer-Verlag Berlin Heidelberg 1999

The idea is to keep a separate table of solutions to subproblems, modify recursive calls to first look up in the table, and then, if the subproblem has been computed, use the saved result, otherwise, compute it and save the result in the table. This method has two advantages. First, the original recursive program needs virtually no change. The underlying interpretation mechanism takes care of the table filling and lookup. Second, only values needed by the original program are actually computed, which is optimal in a sense. Memoization has two disadvantages. First, the mechanism for table filling and lookup has an interpretive overhead. Second, no general strategy for table management is efficient for all problems.

Tabulation determines what shape of table is needed to store the values of all possibly needed subcomputations, introduces appropriate data structures for the table, and computes the table entries in a bottom-up fashion so that the solution to a superproblem is computed using available solutions to subproblems [5,13,40,39,10,12,41,42,21,11]. This overcomes both disadvantages of memoization. First, table filling and lookup are compiled into the resulting program so no separate mechanism is needed for the execution. Second, strategies for table filling and lookup can be specialized to be efficient for particular problems. However, tabulation has two drawbacks. First, it usually requires a thorough understanding of the problem and a complete manual rewrite of the program [14]. Second, to statically ensure that all values possibly needed are computed and stored, a table that is larger than necessary is often used; it may also include solutions to subproblems not actually needed in the original computation.

This paper presents a powerful method that statically analyzes and transforms straightforward recursive programs to efficiently cache and use the results of needed subproblems at appropriate program points in appropriate data structures. The method has three steps: (1) extend the original program to cache all possibly computed values, (2) incrementalize the extended program, with respect to an input increment, to use and maintain all cached results, (3) prune out cached results that are not used in the incremental computation, and finally use the resulting incremental program to form an optimized program. The method overcomes both drawbacks of tabulation. First, it consists of static program analyses and transformations that are general and automatable. Second, it stores only values that are necessary for the optimization; it also shows exactly when and where subproblems not in the original computation are necessarily included.

Our method is based on static analyses and transformations studied previously by others [52,9,48,6,36,20,49,41] and ourselves [33,32,31,27,32] and improves them. Yet, all three steps are simple, automatable, and efficient and have been implemented in a prototype system, CACHET. The system has been used to optimize many programs written as straightforward recursions, including all dynamic programming problems found in [2,46,14]. Performance measurements confirm drastic asymptotic speedups.

2 Formulating the problem

Straightforward solutions to many combinatorics and optimization problems can be written as simple recursions [46,14]. For example, the matrix-chain-multiplication problem [14, pages 302-314] computes the minimum number of

scalar multiplications needed by any parenthesization in multiplying a chain of n matrices, where matrix i has dimensions $p_{i-1} \times p_i$. This can be computed as $m(1, n)$, where $m(i, j)$ computes the minimum number of scalar multiplications for multiplying matrices i through j and can be defined as: for $i \leq j$,

$$m(i,j) = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k \leq j-1}\{m(i,k) + m(k+1,j) + p_{i-1} * p_k * p_j\} & \text{otherwise} \end{cases}$$

The longest-common-subsequence problem [14, pages 314–320] computes the length $c(n, m)$ of the longest common subsequence of two sequences $\langle x_1, ..., x_n \rangle$ and $\langle y_1, y_2, ..., y_m \rangle$, where $c(i, j)$ can be defined as: for $i, j \geq 0$,

$$c(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c(i-1,j-1)+1 & \text{if } i \neq 0 \text{ and } j \neq 0 \text{ and } x_i = y_j \\ \max(c(i,j-1), c(i-1,j)) & \text{otherwise} \end{cases}$$

Both of these examples are literally copied from the textbook by Cormen, Leiserson, and Rivest [14].

These recursive functions can be written straightforwardly in the following first-order, call-by-value functional programming language. A program is a function f_0 defined by a set of mutually recursive functions of the form

$$f(v_1, ..., v_n) \triangleq e$$

where an expression e is given by the grammar

$e ::= v$		variable
$\mid$	$c(e_1, ..., e_n)$	constructor application
$\mid$	$p(e_1, ..., e_n)$	primitive function application
$\mid$	$f(e_1, ..., e_n)$	function application
$\mid$	**if** e_1 **then** e_2 **else** e_3	conditional expression
$\mid$	**let** $v = e_1$ **in** e_2	binding expression

We include arrays as variables and use them for indexed access such as x_i and p_j above. For convenience, we allow global variables to be implicit parameters to functions; such variables can be identified easily for our language even if they are given as explicit parameters. Fig. 1 gives programs for the examples above. Invariants about an input are not part of a program but are written explicitly to be used by the transformations. These examples do not use data constructors, but our previous papers contain a number of examples that use them [33,32,31] and our method handles them.

These straightforward programs repeatedly solve common subproblems and take exponential time. We transform them into dynamic programming algorithms that perform efficient caching and take polynomial time.

We use an asymptotic cost model for measuring time complexity. Assuming that all primitive functions take constant time, we need to consider only values of function applications as candidates for caching. Caching takes extra space, which reflects the well-known trade-off between time and space. Our primary goal is to improve the asymptotic running time of the program. Our secondary goal is to save space by caching only values useful for achieving the primary goal.

Caching requires appropriate data structures. In Step 1, we cache all possibly computed results in a recursive tree following the structure of recursive calls.

$$
\begin{array}{l}
c(i,j) \qquad \text{where } i,j \geq 0 \\
\triangleq \ \textbf{if } i = 0 \ \vee j = 0 \textbf{ then } 0 \\
\quad \textbf{else if } x[i] = y[j] \textbf{ then } c(i-1, j-1) + 1 \\
\quad \textbf{else } \max(c(i, j-1), c(i-1, j))
\end{array}
$$

$$
\begin{array}{l}
m(i,j) \qquad \text{where } i \leq j \\
\triangleq \ \textbf{if } i = j \textbf{ then } 0 \\
\quad \textbf{else } msub(i,j,i)
\end{array}
\qquad
\begin{array}{l}
msub(i,j,k) \qquad \text{where } i \leq k \leq j-1 \\
\triangleq \ \textbf{let } s = m(i,k) + m(k{+}1, j) + p[i{-}1] * p[k] * p[j] \textbf{ in} \\
\quad \textbf{if } k{+}1 = j \textbf{ then } s \\
\quad \textbf{else } \min(s, msub(i,j,k+1))
\end{array}
$$

Fig. 1. Example programs.

Each node of the tree is a tuple that bundles recursive subtrees with the return value of the current call. We use <> to denote a tuple, and we use selectors $1st$, $2nd$, $3rd$, etc. to select the first, second, third, etc. elements of a tuple.

In Step 2, cached values are used and maintained in efficiently computing function calls on slightly incremented inputs. We use an infix operation $\oplus$ to denote an input increment operation, also called an input change (or update) operation. It combines a previous input $x = \langle x_1, ..., x_n \rangle$ and an increment parameter $y = \langle y_1, ..., y_m \rangle$ to form an incremented input $x' = \langle x'_1, ..., x'_n \rangle = x \oplus y$, where each x'_i is some function of x_j's and y_k's. An input increment operation we use for program optimization always has a corresponding decrement operation $prev$ such that for all x, y, and x', if $x' = x \oplus y$ then $x = prev(x')$. Note that y need not be used. For example, an input increment operation to function m in Fig. 1 could be $\langle x'_1, x'_2 \rangle = \langle x_1, x_2 + 1 \rangle$ or $\langle x'_1, x'_2 \rangle = \langle x_1 - 1, x_2 \rangle$, and the corresponding decrement operations are $\langle x_1, x_2 \rangle = \langle x'_1, x'_2 - 1 \rangle$ and $\langle x_1, x_2 \rangle = \langle x'_1 + 1, x'_2 \rangle$, respectively. An input increment to a function that takes a list could be $x' = cons(y, x)$, and the corresponding decrement operation is $x = cdr(x')$.

In Step 3, cached values that are not used for an incremental computation are pruned away, yielding functions that cache, use, and maintain only useful values. Finally, the resulting incremental program is used to form an optimized program. Our optimization preserves the semantics in the sense that if the original program terminates with a values, the optimized program terminates with the same value.

For a function f in an original program, $\bar{f}$ denotes the function that caches all possibly computed values of f, and $\hat{f}$ denotes the pruned function that caches only useful values. We use x to denote an un-incremented input and use r, $\bar{r}$, and $\hat{r}$ to denote the return values of $f(x)$, $\bar{f}(x)$, and $\hat{f}(x)$, respectively. For any function g, we use g' to denote the incremental function that computes $g(x')$, where $x' = x \oplus y$, using cached results about x such as $g(x)$. So, g' may take parameter x', as well as extra parameters each corresponding to a cached result. Fig. 2 summarizes the notation.

3 Step 1: Caching all possibly computed values

Consider a function f_0 defined by a set of recursive functions. Program f_0 may use global variables, such as x and y in function $c(i, j)$. A *possibly computed value* is the value of a function call that is computed for some but not necessarily all

Function	Return Value	Denoted as	Incremental Function
f	original value	r	f'
$\bar{f}$	all possibly computed values	$\bar{r}$	$\bar{f}'$
$\hat{f}$	useful values	$\hat{r}$	$\hat{f}'$

Fig. 2. Notation.

values of the global variables. For example, function $c(i, j)$ computes the value of $c(i-1, j-1)$ only when $x[i] = y[j]$. Such values occur exactly in branches of conditional expressions whose conditions depend on any global variable.

We construct a program $\bar{f}_0$ that caches all possibly computed values in f_0. For example, we extend $c(i, j)$ to always compute the value of $c(i-1, j-1)$ regardless of whether $x[i] = y[j]$. We first apply a simple *hoisting transformation* to lift function calls out of conditional expressions whose conditions depend on global variables. We then apply an *extension transformation* to cache all intermediate results, i.e., values of all function calls, in the return value.

Hoisting transformation. Hoisting transformation $\mathcal{H}st$ identifies conditional expressions whose condition depends on any global variable and then applies the transformation

$$\begin{aligned}\mathcal{H}st[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!] = \ & \textbf{let } v_2 = e_2 \textbf{ in} \\ & \textbf{let } v_3 = e_3 \textbf{ in} \\ & \textbf{if } e_1 \textbf{ then } v_2 \textbf{ else } v_3\end{aligned}$$

For example, the hoisting transformation leaves m and $msub$ unchanged and transforms c into

$$\begin{aligned}c(i, j) \triangleq \ & \textbf{if } i = 0 \ \vee j = 0 \textbf{ then } 0 \\ & \textbf{else let } u_1 = c(i-1, j-1) + 1 \textbf{ in} \\ & \quad \textbf{let } u_2 = \max(c(i, j-1), c(i-1, j)) \textbf{ in} \\ & \quad \textbf{if } x[i] = y[j] \textbf{ then } u_1 \textbf{ else } u_2\end{aligned}$$

$\mathcal{H}st$ simply lifts up the entire subexpressions in the two branches, not just the function calls in them. Administrative simplification performed at the end of the extension transformation will unwind bindings for computations that are used at most once in subsequent computations; thus computations other than function calls will be put down into the appropriate branches then. $\mathcal{H}st$ is simple and efficient. The resulting program has essentially the same size as the original program, so $\mathcal{H}st$ does not increase the running time of the extension transformation or the running times of the later incrementalization and pruning.

If we apply the hoisting transformation on arbitrary conditional expressions, the resulting program may run slower, become non-terminating, or have errors introduced. For conditional expressions whose conditions depend on global variables, we assume that both branches may be executed to terminate correctly regardless of the condition, which holds for the large class of combinatorics and optimization problems we handle. By limiting the hoisting transformation on these conditional expressions, we eliminated the last two problems. The first problem is discussed in Section 6.

Extension transformation. For each hoisted function definition $f(v_1, ..., v_n) \triangleq e$, we construct a function definition

$$\bar{f}(v_1, ..., v_n) \triangleq \mathcal{E}xt[\![e]\!] \tag{1}$$

where $\mathcal{E}xt[\![e]\!]$, defined in [32], extends an expression e to return a nested tuple that contains the values of all function calls made in computing e, i.e., it examines subexpressions of e in applicative order, introduces bindings that name the results of function calls, builds up tuples of these values together with the values of the original subexpressions, and passes these values from subcomputations to enclosing computations. The first component of a tuple corresponds to an original return value. Next, administrative simplifications clean up the resulting program. This yields a program $\bar{f}_0$ that embeds values of all possibly computed function calls in its return value. For the hoisted programs m and c, the extension transformation produces the following functions:

$$\begin{array}{l}
\overline{m}(i,j) \triangleq \ \textbf{if } i = j \textbf{ then } < 0 > \\
\qquad\qquad\ \ \textbf{else } \overline{msub}(i,j,i) \\
\overline{msub}(i,j,k) \triangleq \ \textbf{let } v_1 = \overline{m}(i,k) \textbf{ in} \\
\qquad\qquad\qquad\ \ \textbf{let } v_2 = \overline{m}(k+1,j) \textbf{ in} \\
\qquad\qquad\qquad\ \ \textbf{let } s = 1st(v_1) + 1st(v_2) + p[i-1] * p[k] * p[j] \textbf{ in} \\
\qquad\qquad\qquad\ \ \textbf{if } k+1 = j \textbf{ then } < s, v_1, v_2 > \\
\qquad\qquad\qquad\ \ \textbf{else let } v = \overline{msub}(i,j,k+1) \textbf{ in} \\
\qquad\qquad\qquad\qquad\ \ < \min(s, 1st(v)), v_1, v_2, v > \\
\\
\bar{c}(i,j) \triangleq \ \textbf{if } i = 0 \ \vee j = 0 \textbf{ then } < 0 > \\
\qquad\qquad\ \ \textbf{else let } v_1 = \bar{c}(i-1, j-1) \textbf{ in} \\
\qquad\qquad\qquad\ \ \textbf{let } v_2 = \bar{c}(i, j-1) \textbf{ in} \\
\qquad\qquad\qquad\ \ \textbf{let } v_3 = \bar{c}(i-1, j) \textbf{ in} \\
\qquad\qquad\qquad\ \ \textbf{if } x[i] = y[j] \textbf{ then } < 1st(v_1) + 1, v_1, v_2, v_3 > \\
\qquad\qquad\qquad\ \ \textbf{else } < \max(1st(v_2), 1st(v_3)), v_1, v_2, v_3 >
\end{array}$$

4 Step 2: Static incrementalization

The essence of our method is to use and maintain cached values efficiently as a computation proceeds, i.e., we incrementalize $\bar{f}_0$ with respect to an input increment operation $\oplus$. Precisely, we transform $\bar{f}_0(x \oplus y)$ to use the cached value of $\bar{f}_0(x)$ rather than compute from scratch.

An *input increment operation* $\oplus$ corresponds to a minimal update to the input parameters. We first describe a general method for identifying $\oplus$. We then give a powerful method, called *static incrementalization*, that constructs an incremental version $\bar{f}'$ for each function $\bar{f}$ in the extended program and allows an incremental function to have multiple parameters that represent cached values.

Input increment operation. An input increment should reflect how a computation proceeds. In general, a function may have multiple ways of proceeding depending on the particular computations involved. There is no general method for identifying all of them or the most appropriate ones. Here we propose a method that can systematically identify a general class of them. The idea is to use a *minimal* input change that is in the *opposite* direction of change compared

to arguments of recursive calls. Using the opposite direction of change yields an increment; using a minimal change allows maximum reuse, i.e., maximum incrementality.

Consider a recursively defined function f_0. Formulas for the possible arguments of recursive calls to f_0 in computing $f_0(x)$ can be determined statically. For example, for function $c(i, j)$, recursive calls to c have the set of possible arguments $S_c = \{\langle i-1, j-1\rangle, \langle i, j-1\rangle, \langle i-1, j\rangle\}$, and for function $m(i, j)$, recursive calls to m have the set of possible arguments $S_m = \{\langle i, k\rangle, \langle k+1, j\rangle \mid i \leq k \leq j-1\}$. The latter is simplified from $S_m = \{\langle a, c\rangle, \langle c+1, b\rangle \mid a \leq c \leq b-1, a = i, b = j\}$ where a, b, c are fresh variables that correspond to i, j, k in $msub$; the equalities are based on arguments of the recursive calls involved (in this case $msub$); and the inequalities are obtained from the inequalities on these arguments. The simplification here, as well as the manipulations below, can be done automatically using Omega [44].

Represent the arguments of recursive calls so that the differences between them and x are explicit. For function c, S_c is already in this form, and for function m, S_m is rewritten as $\{\langle i, j-l\rangle, \langle i+l, j\rangle \mid 1 \leq l \leq j-i\}$. Then, extract minimal differences that cover all of these recursive calls. The partial ordering on differences is: a difference involving fewer parameters is smaller; a difference in one parameter with smaller magnitude is smaller; other differences are incomparable. A set of differences *covers* a recursive call if the argument to the call can be obtained by repeated application of the given differences. So, we first compute the set of minimal differences and then remove from it each element that is covered by the remaining elements. For function c, we obtain $\{\langle i, j-1\rangle, \langle i-1, j\rangle\}$, and for function m, we obtain $\{\langle i, j-1\rangle, \langle i+1, j\rangle\}$. Elements of this set represent decrement operations. Finally, take the opposite of each decrement operation to obtain an increment operation $\oplus$, introducing a parameter y if needed (e.g., for increments that use data constructions). For function c, we obtain $\langle i, j+1\rangle$ and $\langle i+1, j\rangle$, and for function m, we obtain $\langle i, j+1\rangle$ and $\langle i-1, j\rangle$. Even though finding input increment operations is theoretically hard in general (and a decrement operation might not have an inverse, in which case our algorithm does not apply), it is usually straightforward.

Typically, a function involves repeatedly solving common subproblems when it contains multiple recursive calls to itself. If there are multiple input increment operations, then any one may be used to incrementalize the program and finally form an optimized program; the rest may be used to further incrementalize the resulting optimized program, if it still involves repeatedly solving common subproblems. For example, for program c, either $\langle i, j+1\rangle$ or $\langle i+1, j\rangle$ will lead to a final optimized program, and for program m, both $\langle i-1, j\rangle$ and $\langle i, j+1\rangle$ need to be used, and they may be used in either order.

Static incrementalization. Given a program $\bar{f}_0$ and an input increment operation $\oplus$, incrementalization symbolically transforms $\bar{f}_0(x')$ for $x' = x \oplus y$ to replace subcomputations with retrievals of their values from the value $\bar{r}$ of $\bar{f}_0(x)$. This exploits equality reasoning, based on control and data structures of the program and properties of primitive operations. The resulting program $\bar{f}_0'$ uses $\bar{r}$ or parts

of $\bar{r}$ as additional arguments, called *cache arguments*, and satisfies: if $\bar{f}_0(x) = \bar{r}$ and $\bar{f}_0(x') = \bar{r}'$, then $\bar{f}_0'(x', \bar{r}) = \bar{r}'$.[1]

The idea is to establish the strongest invariants, especially those about cache arguments, at all calls and maximize their usage. At the end, unused candidate cache arguments are eliminated. Reducing running time corresponds to maximizing uses of invariants; reducing space corresponds to maintaining weakest invariants for all uses. It is important that the methods for establishing and using invariants are specialized so that they are automatable. The precise algorithm is described below. Its use is illustrated afterwards using the running examples.

The algorithm starts with transforming $\bar{f}_0(x')$ for $x' = x \oplus y$ and $\bar{f}_0(x) = \bar{r}$ and first uses the decrement operation to establish an invariant about function arguments. More precisely, it starts with transforming $\bar{f}_0(x')$ with invariant $\bar{f}_0(prev(x')) = \bar{r}$, where $\bar{r}$ is a candidate cache argument. It may use other invariants about x' if given. Invariants given or formed from the enclosing conditions and bindings are called *context*. The algorithm transforms function applications recursively. There are four cases at a function application $f(e_1', ..., e_n')$.

(1) If $f(e_1', ..., e_n')$ specializes, by definition of f, under its context to a base case, i.e., an expression with no recursive calls, then replace it with the specialized expression.
(2) Otherwise, if $f(e_1', ..., e_n')$ equals a retrieval from a cache argument based on an invariant about the cache argument in its context, then replace it with the retrieval.
(3) Otherwise, if an incremental version f' of f has been introduced, then replace $f(e_1', ..., e_n')$ with a call to f' if the corresponding invariants can be maintained; if some invariants can not be maintained, then eliminate them and retransform from where f' was introduced.
(4) Otherwise, introduce an incremental version f' of f and replace $f(e_1', ..., e_n')$ with a call to f', as described below.

In general, the replacement in case (1) is also done, repeatedly, if the specialized expression contains only recursive calls whose arguments are closer to, and will equal after a bounded number of such replacements, arguments for base cases or arguments on which retrievals can be done. Since a bounded number of invariants are used at a function application, as described below, the retransformation in case (3) can only be done a bounded number of times. So, the algorithm always terminates.

To introduce an incremental version f' of f at $f(e_1', ..., e_n')$, let *Inv* be the set of invariants about cache arguments or context information at $f(e_1', ..., e_n')$. Those about cache arguments are of the form $g_i(e_{i1}, ..., e_{in_i}) = e_{ir}$, where e_{ir} is either a candidate cache argument in the enclosing environment or a selector applied to such an argument. Those about context information are of the form $e = true$, $e = false$, or $v = e$, obtained from conditions or bindings. For simplicity, we assume that all bound variables are renamed so that they are distinct. Introduce f' to compute $f(x_1'', ..., x_n'')$ for $x_1'' = e_1', ..., x_n'' = e_n'$, where $x_1'', ..., x_n''$ are fresh variables, and deduce invariants about $x_1'', ..., x_n''$ based on *Inv*. The deduction uses equations $e_1' = x_1'', ..., e_n' = x_n''$ to eliminate variables in *Inv* and can

[1] In previous papers, we defined $\bar{f}_0'$ slightly differently: if $\bar{f}_0(x) = \bar{r}$ and $\bar{f}_0(x \oplus y) = \bar{r}'$, then $\bar{f}_0'(x, y, \bar{r}) = \bar{r}'$.

be done automatically using Omega [44]. Resulting equations relating $x''_1, ..., x''_n$ are used also to duplicate other invariants deduced. If a resulting invariant still uses a variable other than $x''_1, ..., x''_n$, discard it. Finally, for each invariant about a cache argument, replace its right hand side with a fresh variable, which becomes a candidate cache argument of f'. This yields the set of invariants now associated with f'. Note that invariants about cache arguments have the form $g_i(e''_{i1}, ..., e''_{in_i}) = r_i$, where $e''_{i1}, ..., e''_{in_i}$ use only variables $x''_1, ..., x''_n$, and r_i is a fresh variable. Among the left hand sides of these invariants, identify an application of f whose arguments have a minimum difference from $x''_1, ..., x''_n$; if such an application exists, denote it $f(e''_1, ..., e''_n)$.

To obtain a definition of f', unfold $f(x''_1, ..., x''_n)$ and then exploit conditionals in $f(x''_1, ..., x''_n)$ and $f(e''_1, ..., e''_n)$ (if it exists) and components in the candidate cache arguments of f'. To exploit conditionals in $f(x''_1, ..., x''_n)$, move function applications inside branches of the conditionals in $f(x''_1, ..., x''_n)$ whenever possible, preserving control dependencies incurred by the order of conditional tests and data dependencies incurred by the bindings. This is done by repeatedly applying the following transformation in applicative order to the unfolded expression. For any $t(e_1, ..., e_k)$ being $c(e_1, ..., e_k)$, $p(e_1, ..., e_k)$, $f(e_1, ..., e_k)$, **if** e_1 **then** e_2 **else** e_3, or **let** $v = e_1$ **in** e_2, if e_i is **if** e_{i1} **then** e_{i2} **else** e_{i3}, where $i \neq 2, 3$ if t is a conditional, and $i \neq 2$ or e_{i1} does not depend on v if t is a binding expression, then transform $t(e_1, ..., e_k)$ to **if** e_{i1} **then** $t(e_1, ..., e_{i-1}, e_{i2}, e_{i+1}, ..., e_k)$ **else** $t(e_1, ..., e_{i-1}, e_{i3}, e_{i+1}, ..., e_k)$. This transformation preserves the semantics. It may increase the code size, but it does not increase the running time of the resulting program. To exploit the conditionals in $f(e''_1, ..., e''_n)$, introduce conditions from $f(e''_1, ..., e''_n)$ in the transformed expression just obtained and put function applications inside both branches that follow such a condition. This is done by applying the following transformation in outermost-first order to the conditionals in the transformed expression just obtained. For each branch e_i of the conditional that contains a function application, let e be the outermost condition in $f(e''_1, ..., e''_n)$ that is not implied by the context of e_i; if e uses only variables defined in the context of e_i and takes constant time to compute, and the two branches in $f(e''_1, ..., e''_n)$ that depend on e contain different function applications in some component, then transform e_i to **if** e **then** e_i **else** e_i. To exploit each component in a candidate cache argument r_i where there is an invariant $g_i(e''_{i1}, ..., e''_{in_i}) = r_i$, for each branch in the transformed expression, specialize $g_i(e''_{i1}, ..., e''_{in_i})$ under the context of that branch. This may yield additional function applications that equal various components of r_i. After these control structures and data structures are exploited, we simplify primitive operations on $x'_1, ..., x'_n$ and transform function applications recursively based on the four cases described. Finally, after we obtain a definition of f', replace the function application $f(e'_1, ..., e'_n)$ with a call to f' with arguments $e'_1, ..., e'_n$ and cache arguments e_{ir}'s for the invariants used.

The simplifications and equality reasoning needed for all the problems we have encountered involve only recursive data structures and Presburger arithmetic and can be fully automated.

Longest common subsequence. Incrementalize c under $\langle i', j' \rangle = \langle i + 1, j \rangle$. We start with $\bar{c}(i', j')$, with cache argument $\bar{r}$ and invariant $\bar{c}(prev(i', j')) = \bar{c}(i' - 1, j') = \bar{r}$; the invariants $i', j' > 0$ may also be included but do not affect any

transformation below, so they are omitted for convenience. This is case (4), so we introduce incremental version $\bar{c}'$ to compute $\bar{c}(i', j')$. Unfolding the definition of $\bar{c}$ and listing conditions to be exploited, we obtain the code below. The false branch of $\bar{c}(i', j')$ is duplicated with the additional condition $i' - 1 = 0 \ \vee j' = 0$, which is copied from the condition in definition of $\bar{c}(i' - 1, j')$; for convenience, three function applications bounded to v_1 to v_3 are not put inside branches that follow condition $x[i'] = y[j']$, since their transformations are not affected, and simplification at the end can take them back out.

$$
\begin{array}{l}
\bar{c}(i', j') = \ \textbf{if } i' = 0 \ \vee j' = 0 \textbf{ then } < 0 > \\
\qquad\qquad \textbf{else if } i' - 1 = 0 \ \vee j' = 0 \textbf{ then} \\
\qquad\qquad\qquad \textbf{let } v_1 = \bar{c}(i' - 1, j' - 1) \textbf{ in} \\
\qquad\qquad\qquad \textbf{let } v_2 = \bar{c}(i', j' - 1) \textbf{ in} \\
\qquad\qquad\qquad \textbf{let } v_3 = \bar{c}(i' - 1, j') \textbf{ in} \\
\qquad\qquad\qquad \textbf{if } x[i'] = y[j'] \textbf{ then } < 1st(v_1) + 1, v_1, v_2, v_3 > \\
\qquad\qquad\qquad \textbf{else } < \max(1st(v_2), 1st(v_3)), v_1, v_2, v_3 > \\
\qquad\qquad \textbf{else let } v_1 = \bar{c}(i' - 1, j' - 1) \textbf{ in} \\
\qquad\qquad\qquad \textbf{let } v_2 = \bar{c}(i', j' - 1) \textbf{ in} \\
\qquad\qquad\qquad \textbf{let } v_3 = \bar{c}(i' - 1, j') \textbf{ in} \\
\qquad\qquad\qquad \textbf{if } x[i'] = y[j'] \textbf{ then } < 1st(v_1) + 1, v_1, v_2, v_3 > \\
\qquad\qquad\qquad \textbf{else } < \max(1st(v_2), 1st(v_3)), v_1, v_2, v_3 >
\end{array}
$$

In the second branch, $i' - 1 = 0$ is true, since $j' = 0$ would imply that the first branch is taken. The first and third calls fall in case (1) and specialize to $< 0 >$. The second call falls in case (3) and equals a recursive call to $\bar{c}'$ with arguments $i', j' - 1$ and cache argument $< 0 >$ since we have a corresponding invariant $\bar{c}(i' - 1, j' - 1) = < 0 >$. Additional simplification unwinds bindings for v_1 and v_3, simplifies $1st(< 0 >) + 1$ to 1, and simplifies $\max(1st(v_2), 1st(< 0 >))$ to $1st(v_2)$.

In the third branch, condition $i' - 1 = 0 \vee j' = 0$ is false; $\bar{c}(i' - 1, j')$ by definition of $\bar{c}$ equals its second branch where $\bar{c}(i' - 1, j' - 1)$ is bound to v_2, and thus $\bar{c}(i' - 1, j') = \bar{r}$ implies $\bar{c}(i' - 1, j' - 1) = 3rd(\bar{r})$. The first call falls in case (2) and equals $3rd(\bar{r})$. The second call falls in case (3) and equals a recursive call to $\bar{c}'$ with arguments $i', j' - 1$ and cache argument $3rd(\bar{r})$) since we have a corresponding invariant $\bar{c}(i' - 1, j' - 1) = 3rd(\bar{r})$. The third call falls in case (2) and equals $\bar{r}$. We obtain

$$
\begin{array}{l}
\bar{c}'(i', j', \bar{r}) \triangleq \ \textbf{if } i' = 0 \ \vee j' = 0 \textbf{ then } < 0 > \\
\qquad\qquad \textbf{else if } i' - 1 = 0 \textbf{ then} \\
\qquad\qquad\qquad \textbf{let } v_2 = \bar{c}'(i', j' - 1, < 0 >) \textbf{ in} \\
\qquad\qquad\qquad \textbf{if } x[i'] = y[j'] \textbf{ then } < 1, < 0 >, v_2, < 0 >> \\
\qquad\qquad\qquad \textbf{else } < 1st(v_2), < 0 >, v_2, < 0 >> \\
\qquad\qquad \textbf{else let } v_1 = 3rd(\bar{r}) \textbf{ in} \\
\qquad\qquad\qquad \textbf{let } v_2 = \bar{c}'(i', j' - 1, 3rd(\bar{r})) \textbf{ in} \\
\qquad\qquad\qquad \textbf{let } v_3 = \bar{r} \textbf{ in} \\
\qquad\qquad\qquad \textbf{if } x[i'] = y[j'] \textbf{ then } < 1st(v_1) + 1, v_1, v_2, v_3 > \\
\qquad\qquad\qquad \textbf{else } < \max(1st(v_2), 1st(v_3)), v_1, v_2, v_3 >
\end{array}
$$

If $\bar{r} = \bar{c}(i' - 1, j')$, then $\bar{c}'(i', j', \bar{r}) = \bar{c}(i', j')$, and $\bar{c}'$ takes time and space linear in j', for caching and maintaining a linear list.

Matrix-chain multiplication. Incrementalize m under $\langle i', j' \rangle = \langle i, j+1 \rangle$. We start with $\overline{m}(i', j')$, with cache argument $\bar{r}$ and invariants $\overline{m}(i', j' - 1) = \bar{r}$ and $i' \leq j'$.

This is case (4), so we introduce incremental version $\overline{m}'$ to compute $\overline{m}(i', j')$. Unfolding $\overline{m}$, listing conditions, and specializing the second branch, we obtain the code below.

$$\begin{array}{l}\overline{m}(i', j') = \ \textbf{if } i' = j' \textbf{ then } < 0 > \\ \qquad\qquad \textbf{else if } i' = j' - 1 \textbf{ then } < p[i' - 1] * p[i'] * p[j'], < 0 >, < 0 >> \\ \qquad\qquad \textbf{else } \overline{msub}(i', j', i')\end{array}$$

In the third branch, condition $i' = j' - 1$ is false; $\overline{m}(i', j' - 1)$ by definition of $\overline{m}$ equals $\overline{msub}(i', j' - 1, i')$, and thus $\overline{m}(i', j' - 1) = \bar{r}$ implies $\overline{msub}(i', j' - 1, i') = \bar{r}$. The call $\overline{msub}(i', j', i')$ falls in case (4). We introduce $\overline{msub}'$ to compute $\overline{msub}(i'', j'', k'')$ for $i'' = i', j'' = j', k'' = i'$, with invariants $\overline{msub}(i', j' - 1, i') = \bar{r}$, $\overline{m}(i', j' - 1) = \bar{r}$, $i' \leq j'$, $i' \neq j'$, $i' \neq j' - 1$. Express these invariants as invariants on i'', j'', k'' using Omega, and introduce fresh variables $\bar{r_i}$ for candidate cache arguments. We obtain

$$\begin{array}{l}\overline{msub}(i'', j'' - 1, k'') = \bar{r_1},\ \overline{m}(i'', j'' - 1) = \bar{r_2},\ i'' \leq j'',\ i'' \neq j'',\ i'' \neq j'' - 1,\ k'' = i'', \\ \overline{msub}(i'', j'' - 1, i'') = \bar{r_3}, \qquad\qquad\qquad\qquad\qquad\qquad k'' \leq j'',\ k'' \neq j'',\ k'' \neq j'' - 1, \\ \overline{msub}(k'', j'' - 1, k'') = \bar{r_4},\ \overline{m}(k'', j'' - 1) = \bar{r_5}, \\ \overline{msub}(k'', j'' - 1, i'') = \bar{r_6},\end{array} \tag{2}$$

where equation $k'' = i''$ is an additional invariant deduced, and invariants not on the first line are duplications of those in the first line based on $k'' = i''$. Arguments of $\overline{msub}(i'', j'' - 1, k'')$ have a minimum difference from arguments of $\overline{msub}(i'', j'', k'')$.

Unfolding $\overline{msub}(i'', j'', k'')$ and listing conditions to be exploited, we obtain the following code. The code for v_1 and v_2 is duplicated for both branches that follow the condition $k'' + 1 = j''$. The code for v is duplicated for both branches that follow the additional condition $k'' + 1 = j'' - 1$, which is copied from the condition in the definition of $\overline{msub}(i'', j'' - 1, k'')$.

$$\begin{array}{l}\overline{msub}(i'', j'', k'') = \ \textbf{if } k'' + 1 = j'' \textbf{ then} \\ \qquad\qquad\qquad\qquad \textbf{let } v_1 = \overline{m}(i'', k'') \textbf{ in} \\ \qquad\qquad\qquad\qquad \textbf{let } v_2 = \overline{m}(k'' + 1, j'') \textbf{ in} \\ \qquad\qquad\qquad\qquad \textbf{let } s = 1st(v_1) + 1st(v_2) + p[i'' - 1] * p[k''] * p[j''] \textbf{ in} \\ \qquad\qquad\qquad\qquad < s, v_1, v_2 > \\ \qquad\qquad\qquad \textbf{else let } v_1 = \overline{m}(i'', k'') \textbf{ in} \\ \qquad\qquad\qquad\qquad \textbf{let } v_2 = \overline{m}(k'' + 1, j'') \textbf{ in} \\ \qquad\qquad\qquad\qquad \textbf{let } s = 1st(v_1) + 1st(v_2) + p[i'' - 1] * p[k''] * p[j''] \textbf{ in} \\ \qquad\qquad\qquad\qquad \textbf{if } k'' + 1 = j'' - 1 \textbf{ then} \\ \qquad\qquad\qquad\qquad\qquad \textbf{let } v = \overline{msub}(i'', j'', k'' + 1) \textbf{ in} \\ \qquad\qquad\qquad\qquad\qquad < \min(s, 1st(v)), v_1, v_2, v > \\ \qquad\qquad\qquad\qquad \textbf{else let } v = \overline{msub}(i'', j'', k'' + 1) \textbf{ in} \\ \qquad\qquad\qquad\qquad\qquad < \min(s, 1st(v)), v_1, v_2, v >\end{array}$$

The first branch is simplified away since we have invariant $k'' \neq j'' - 1$.

In the other branch, $\overline{msub}(i'', j'' - 1, k'')$ by definition of $\overline{msub}$ has $\overline{m}(i'', k'')$ bound to v_1 and $\overline{m}(k'' + 1, j'' - 1)$ bound to v_2, and thus $\overline{msub}(i'', j'' - 1, k'') = \bar{r_1}$ implies $\overline{m}(i'', k'') = 2nd(\bar{r_1})$ and $\overline{m}(k'' + 1, j'' - 1) = 3rd(\bar{r_1})$. The first call falls in case (1), since we have invariant $k'' = i''$, and equals $< 0 >$. The second call falls in case (3) and equals a recursive call to $\overline{m}'$ with arguments $k'' + 1, j''$ and cache

argument $3rd(\bar{r_1})$ since we have a corresponding invariant $\overline{m}(k''+1, j''-1) = 3rd(\bar{r_1})$.

In the branch where $k''+1 = j''-1$ is true, the call to $\overline{msub}$ falls in case (1) and equals

$$\begin{array}{l}\textbf{let } v_1 = \overline{m}(i'', j''-1) \textbf{ in let } v_2 = \overline{m}(j'', j'') \textbf{ in}\\ \textbf{let } s = 1st(v_1) + 1st(v_2) + p[i''-1] * p[k''+1] * p[j''] \textbf{ in } < s, v_1, v_2 >\end{array}$$

which then equals $<1st(\bar{r_2}) + p[i''-1] * p[k''+1] * p[j''],\ \bar{r_2}, <0>>$ because the first call equals $\bar{r_2}$ and the second call equals $< 0 >$.

In the last branch, the call to $\overline{msub}$ falls in case (3). However, the arguments of this call do not satisfy the invariant corresponding to $k'' = i''$ and those on the third and fourth lines in (2). So we delete these invariants and retransform $\overline{msub}$. Everything remains the same except that $\overline{m}(i'', k'')$ does not fall in case (1) any more; it falls in case (2) and equals $2nd(\bar{r_1})$. We replace this call to $\overline{msub}$ by a recursive call to $\overline{msub}'$ with arguments $i'', j'', k''+1$ and cache arguments $4th(\bar{r_1})$, $\bar{r_2}$, $\bar{r_3}$ since we have corresponding invariants $\overline{msub}(i'', j''-1, k''+1) = 4th(\bar{r_1})$, $\overline{m}(i'', j''-1) = \bar{r_2}$, $\overline{m}(i'', j''-1, i'') = \bar{r_3}$.

We eliminate unused candidate cache argument $\bar{r_3}$, and we replace the original call $\overline{msub}(i', j', i')$ by $\overline{msub}(i', j', i', \bar{r}, \bar{r})$. We obtain

$$\begin{array}{l}\overline{m}'(i', j', \bar{r}) \triangleq \textbf{ if } i' = j' \textbf{ then } < 0 >\\ \qquad\qquad \textbf{else if } i' = j' - 1 \textbf{ then } < p[i'-1] * p[i'] * p[j'], <0>, <0>>\\ \qquad\qquad \textbf{else } \overline{msub}'(i', j', i', \bar{r}, \bar{r})\end{array}$$

$$\begin{array}{l}\overline{msub}'(i'', j'', k'', \bar{r_1}, \bar{r_2}) \triangleq\\ \qquad \textbf{let } v_1 = 2nd(\bar{r_1}) \textbf{ in}\\ \qquad \textbf{let } v_2 = \overline{m}'(k''+1, j'', 3rd(\bar{r_1})) \textbf{ in}\\ \qquad \textbf{let } s = 1st(v_1) + 1st(v_2) + p[i''-1] * p[k''] * p[j''] \textbf{ in}\\ \qquad \textbf{if } k''+1 = j''-1 \textbf{ then}\\ \qquad\qquad \textbf{let } v = <1st(\bar{r_2}) + p[i''-1] * p[k''+1] * p[j''],\ \bar{r_2}, <0>> \textbf{ in}\\ \qquad\qquad < \min(s, 1st(v)), v_1, v_2, v >\\ \qquad \textbf{else let } v = \overline{msub}'(i'', j'', k''+1, 4th(\bar{r_1}), \bar{r_2}) \textbf{ in}\\ \qquad\qquad < \min(s, 1st(v)), v_1, v_2, v >\end{array}$$

If $\bar{r} = \overline{m}(i', j'-1)$, then $\overline{m}'(i', j', \bar{r}) = \overline{m}(i', j')$, and $\overline{m}'$ is an exponential-factor faster. However, $\overline{m}$ still takes exponential time due to repeated calls to $\overline{m}'$; incrementalizing again under $\langle i', j' \rangle = \langle i-1, j \rangle$, we obtain a linear-time incremental program.

5 Step 3: Pruning unnecessary values

Among the components maintained by $\bar{f_0}'(x', \bar{r})$, the first one is the return value of $f_0(x')$. Components in $\bar{r}$ that are not useful for computing this value need not be cached and maintained. We prune the programs $\bar{f_0}$ and $\bar{f_0}'$ and obtain a program $\hat{f_0}$ that caches only the useful values and a program $\hat{f_0}'$ that uses and maintains only the useful values. Finally, we form an optimized program that computes f_0 by using the base cases in $\hat{f_0}$ and by repeatedly using the incremental version $\hat{f_0}'$.

Pruning. Pruning requires a dependence analysis that can precisely describe substructures of recursive trees [32]. We use an analysis method based on regular tree grammars [28]. We have implemented a simplified version that uses set constraints to efficiently produce precise analysis results. Pruning can save space, as well as time, and reduce code size.

For example, in program $\bar{c}'$, only the third component of $\bar{r}$ is useful. Pruning the second and fourth components of $\bar{c}$ and $\bar{c}'$, which moves the third up to the second, and doing a few simplifications, which transforms $1st(\bar{c})$ back to c and unwinds bindings for v_1 and v_3, we obtain $\hat{c}$ and $\hat{c}'$ below:

$$\begin{array}{l}\hat{c}(i,j) \triangleq \textbf{if } i=0 \ \vee j=0 \textbf{ then } <0> \\ \qquad\qquad \textbf{else let } v_2 = \hat{c}(i,j-1) \textbf{ in} \\ \qquad\qquad\quad \textbf{if } x[i]=y[j] \textbf{ then } < c(i-1,j-1))+1, v_2> \\ \qquad\qquad\quad \textbf{else } < \max(1st(v_2), c(i-1,j))), v_2>\end{array}$$

$$\begin{array}{l}\hat{c}'(i',j',\hat{r}) \triangleq \textbf{if } i'=0 \ \vee j'=0 \textbf{ then } <0> \\ \qquad\qquad \textbf{else if } i'-1=0 \textbf{ then} \\ \qquad\qquad\quad \textbf{let } v_2 = \hat{c}'(i',j'-1,<0>) \textbf{ in} \\ \qquad\qquad\quad \textbf{if } x[i']=y[j'] \textbf{ then } <1, v_2> \\ \qquad\qquad\quad \textbf{else } <1st(v_2), v_2> \\ \qquad\qquad \textbf{else let } v_2 = \hat{c}'(i',j'-1,2nd(\hat{r})) \textbf{ in} \\ \qquad\qquad\quad \textbf{if } x[i']=y[j'] \textbf{ then } <1st(2nd(\hat{r}))+1, v_2> \\ \qquad\qquad\quad \textbf{else } < \max(1st(v_2), 1st(\hat{r})), v_2>\end{array}$$

Pruning leaves programs $\overline{m}$ and $\overline{m}'$ unchanged. We obtain the same programs $\widehat{m}$ and $\widehat{m}'$, respectively.

Forming optimized programs. We redefine functions f_0 and $\hat{f}_0$ and use function $\hat{f}_0'$:

$$\begin{array}{l}f_0(x) \triangleq 1st(\hat{f}_0(x)) \\ \hat{f}_0(x) \triangleq \textbf{if } base\_cond(x) \textbf{ then } base\_val(x) \textbf{ else let } \hat{r} = \hat{f}_0(prev(x)) \textbf{ in } \hat{f}_0'(x,\hat{r})\end{array}$$

where *base_cond* is the base-case condition, and *base_val* is the corresponding value, both copied from the definition of $\hat{f}_0$. In general, there may be multiple base cases, and we just list them all.

For examples c and m, we obtain directly

$$\begin{array}{l}c(i,j) \triangleq 1st(\hat{c}(i,j)) \\ \hat{c}(i,j) \triangleq \textbf{if } i=0 \ \vee j=0 \textbf{ then } <0> \textbf{ else let } \hat{r} = \hat{c}(i-1,j) \textbf{ in } \hat{c}'(i,j,\bar{r})\end{array}$$

$$\begin{array}{l}m(i,j) \triangleq 1st(\widehat{m}(i,j)) \\ \widehat{m}(i,j) \triangleq \textbf{if } i=j \textbf{ then } <0> \textbf{ else let } \hat{r} = \widehat{m}(i,j-1) \textbf{ in } \widehat{m}'(i,j,\bar{r})\end{array}$$

where $\hat{c}'$ and $\widehat{m}'$ are as obtained above. For $c(n,m)$, while the original program takes $O(2^{n+m})$ time, the optimized program takes $O(n*m)$ time. For $m(1,n)$, while the original program takes $O(n*3^n)$ time, the optimized program takes $O(n^2*2^n)$ time. Incrementalizing the optimized program again under the increment to the other parameter, we obtain an optimized program that takes $O(n^3)$ time.

6 Summary and discussion

Our method for dynamic programming is completely static, fully automatable, and efficient. In particular, it is based on a general approach for program optimization—incrementalization. Although our static incrementalization allows only one incremental version for each original function, it is still powerful enough to incrementalize all examples in [33,32,31], including various list manipulations, matrix computations, attribute evaluation, and graph problems. We believe that our method can perform dynamic programming for all problems whose solutions involve recursively solving subproblems that overlap, but a formal justification awaits more rigorous study.

In our method, only values that are necessary for the incrementalization are stored, in appropriate data structures. For the longest-common-subsequence example, only a linear list is needed, whereas in standard textbooks, a quadratic two-dimensional array is used, and an additional optimization is needed to reduce it to a one-dimensional array [14]. For the matrix-chain-multiplication example, our optimized program uses a list of lists that forms a triangle shape, rather than a two-dimensional array of square shape. It's nontrivial to see that recursive data structures gives the same asymptotic speedup as arrays for these examples. There are dynamic programming problems, e.g., 0-1 knapsack, for which the use of array, with constant-time access of elements, helps achieve desired asymptotic speedups. Such situations become evident when doing incrementalization and can be taken care of easily. This will be described in a future paper. Although we present the optimizations for a functional language, the underlying principle is general and has been applied to programs that use loops and arrays [27,30].

Some values computed in a hoisted program might not be computed by the original program and are therefore called *auxiliary information* [31]. Both incrementalization and pruning produce programs that are as least as fast as the given program, but caching auxiliary information may result in a slower program on certain inputs. We can determine statically whether such information is cached in the final program. If so, we can use time and space analysis [29] to determine whether it is worthwhile to use and maintain such information.

Many dynamic programming algorithms can be further improved by exploiting additional properties of the given problems [7], e.g., greedy properties. Our method is not specially aimed at discovering such properties. Nevertheless, it can maintain such properties once they are added. For example, for the paragraph-formatting problem [14,17], we can derive a quadratic-time algorithm that uses dynamic programming; if the original program has a simple extra conditional that follows from a greedy property, our derived dynamic programming program uses it as well and takes linear time with a factor of line width. How to systematically discover and use these additional properties is a subject for future study.

7 Implementation and experimentation results

All three steps have been implemented in a prototype system, CACHET. The incrementalization step as currently implemented is semi-automatic [26] and is being automated. The implementation uses the Synthesizer Generator [47].

Fig. 3 summarizes some of the examples derived (most of them semi-automatically and some automatically) and compares their asymptotic running times.[2] The second column shows whether more than one cache argument is needed in an incremental program. The third column shows whether the incremental program computes values not necessarily computed by the original program. Paragraph formatting 2 [17] includes a conditional that reflects a greedy property. The "a" in the third column for the last two examples shows that cached values are stored in arrays. Performance measurements confirmed drastic speedups.

Examples	multiple cache arg	aux info	original program's running time	optimized prog's running time
Fibonacci function [39]			$O(2^n)$	$O(n)$
binomial coefficients [39]			$O(2^n)$	$O(n*k)$
longest common subsequence [14]		$\surd$	$O(2^{n+m})$	$O(n*m)$
matrix-chain multiplication [14]	$\surd$		$O(n*3^n)$	$O(n^3)$
string editing distance [46]			$O(3^{n+m})$	$O(n*m)$
dag path sequence [6]		$\surd$	$O(2^n)$	$O(n^2)$
optimal polygon triangulation [14]	$\surd$		$O(n*3^n)$	$O(n^3)$
optimal binary search trees [2]	$\surd$		$O(n*3^n)$	$O(n^3)$
paragraph formatting [14]	$\surd$		$O(n*2^n)$	$O(n^2)$
paragraph formatting 2	$\surd$		$O(n*2^n)$	$O(n*width)$
0-1 knapsack [14]		$\surd$a	$O(2^n)$	$O(n*weight)$
context-free-grammar parsing [2]	$\surd$	$\surd$a	$O(n*(2*size+1)^n)$	$O(n^3*size)$

Fig. 3. Summary of Examples.

8 Related work and conclusion

Dynamic programming was first formulated by Bellman [4] and has been studied extensively since [51]. Bird [5], de Moor [16], and others have studied it in the context of program transformation. While some works address the derivation of recursive equations, notably the work by Smith [50], our work addresses the derivation of efficient programs that use tabulation. Previous methods for this problem either apply to specific subclasses of problems [13,40,10,12,42,21] or give general frameworks and strategies rather than precise algorithms [52,9,5,48,6,3,39,49,8], [16,41,15]. Our work is based on the general principle of incrementalization [38,31] and consists of precise program analyses and transformations.

In particular, tupling [40,41] aims to compute multiple values together in an efficient way. It is improved to be automatic on subclasses of problems [10] and to work on more general forms [12]. It is also extended to store lists of values [42], but such lists are generated in a fixed way, which is not the most appropriate way for many programs. A special form of tupling can eliminate multiple data traversals for many functions [21]. A method specialized for introducing arrays was proposed for tabulation [11], but as our method has shown, array is not

[2] Matrix-chain multiplication, optimal binary search trees, optimal polygon triangulation, and other problems not in Fig. 3 have similar control structures for recursive calls. Yet, it is nontrivial for an automated system to handle all of them uniformly.

essential for the speedup of many programs; their arrays are complicated to derive and often consume more space than necessary.

Compared with our previous work for incrementalizing functional programs [33,32,31], this work contains drastic improvements. First, our previous work address the systematic derivation of an incremental program f' given both program f and operation $\oplus$. This paper describes a systematic method for identifying an appropriate operation $\oplus$ given a function f and using the derived incremental program f' to form an optimized version of f. Second, since it is difficult to introduce appropriate cache arguments, our previous method allows at most one cache argument for each incremental function. This paper allows multiple cache arguments, without which many programs could not be incrementalized, e.g., the matrix-chain-multiplication program. Third, our previous method introduces incremental functions using an on-line strategy, i.e., on-the-fly during the transformation, so it may attempt to introduce an unbounded number of new functions and thus not terminate. The algorithm in this paper statically determines one incremental function for each one in the original program, i.e., it is monovariant; even though it is theoretically more limited, it is simpler, always terminates, and is able to incrementalize all previous examples. Finally, based on the idea of cache-and-prune that was proposed earlier [32], the method in this paper uses hoisting to extend the set of intermediate results [32] to include a kind of auxiliary information [31] that is sufficient for dynamic programming. This method is simpler than our previous general method for discovering auxiliary information [31]. Additionally, we now use a more precise and efficient dependence analysis for pruning [28].

Finite differencing [38,37] is based on the same underlying principle as incremental computation. Paige has explicitly asked whether finite differencing can be generalized to handle dynamic programming [36]; it is clear that he perceived an important connection. However, finite differencing has been formulated for set-based languages, while straightforward solutions to dynamic programming problems are usually formulated as recursive functions, so it was difficult to actually establish the connection.

Overall, being able to incrementalize complicated recursion in a systematic way is a more drastic improvement complementing previous methods for incrementalizing loops [38,27]. Our new method based on static incrementalization is general and fully automatable. Based on our existing implementation, we believe that a complete system will perform incrementalization efficiently.

References

1. M. Abadi, B. Lampson, and J.-J. Lévy. Analysis and caching of dependencies. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*. ACM, New York, May 1996.
2. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass., 1974.
3. F. L. Bauer, B. Möller, H. Partsch, and P. Pepper. Formal program construction by transformations—Computer-aided, intuition-guided programming. *IEEE Trans. Softw. Eng.*, 15(2):165–180, Feb. 1989.
4. R. E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, New Jersey, 1957.
5. R. S. Bird. Tabulation techniques for recursive programs. *ACM Comput. Surv.*, 12(4):403–417, Dec. 1980.

6. R. S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Trans. Program. Lang. Syst.*, 6(4):487–504, Oct. 1984.
7. R. S. Bird and O. de Moor. From dynamic programming to greedy algorithms. In B. Möller, H. Partsch, and S. Schuman, editors, *Formal Program Development*, volume 755 of *Lecture Notes in Computer Science*, pages 43–61. Springer-Verlag, Berlin, 1993.
8. E. A. Boiten. Improving recursive functions by inverting the order of evaluation. *Sci. Comput. Program.*, 18(2):139–179, Apr. 1992.
9. R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, Jan. 1977.
10. W.-N. Chin. Towards an automated tupling strategy. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 119–132. ACM, New York, June 1993.
11. W.-N. Chin and M. Hagiya. A bounds inference method for vector-based memoization. In ICFP 1997 [23], pages 176–187.
12. W.-N. Chin and S.-C. Khoo. Tupling functions with multiple recursion parameters. In P. Cousot, M. Falaschi, G. Filè, and A. Rauzy, editors, *Proceedings of the 3rd International Workshop on Static Analysis*, volume 724 of *Lecture Notes in Computer Science*, pages 124–140. Springer-Verlag, Berlin, Sept. 1993.
13. N. H. Cohen. Eliminating redundant recursive calls. *ACM Trans. Program. Lang. Syst.*, 5(3):265–299, July 1983.
14. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press/McGraw-Hill, 1990.
15. S. Curtis. Dynamic programming: A different perspective. In R. Bird and L. Meertens, editors, *Algorithmic Languages and Calculi*, pages 1–23. Chapman & Hall, London, U.K., 1997.
16. O. de Moor. A generic program for sequential decision processes. In M. Hermenegildo and D. S. Swierstra, editors, *Programming Languages: Implementations, Logics, and Programs*, volume 982 of *Lecture Notes in Computer Science*, pages 1–23. Springer-Verlag, Berlin, 1995.
17. O. de Moor and J. Gibbons. Bridging the algorithm gap: A linear-time functional program for paragraph formatting. Technical Report CMS-TR-97-03, School of Computing and Mathematical Sciences, Oxford Brookes University, July 1997.
18. J. Field and T. Teitelbaum. Incremental reduction in the lambda calculus. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 307–322. ACM, New York, June 1990.
19. D. P. Friedman, D. S. Wise, and M. Wand. Recursive programming through table look-up. In *Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation*, pages 85–89. ACM, New York, 1976.
20. Y. Futamura and K. Nogi. Generalized partial evaluation. In B. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 133–151. North-Holland, Amsterdam, 1988.
21. Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling calculation eliminates multiple data traversals. In ICFP 1997 [23], pages 164–175.
22. J. Hughes. Lazy memo-functions. In *Proceedings of the 2nd Conference on Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 129–146. Springer-Verlag, Berlin, Sept. 1985.
23. *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*. ACM, New York, June 1997.
24. R. M. Keller and M. R. Sleep. Applicative caching. *ACM Trans. Program. Lang. Syst.*, 8(1):88–108, Jan. 1986.
25. H. Khoshnevisan. Efficient memo-table management strategies. *Acta Informatica*, 28(1):43–81, 1990.
26. Y. A. Liu. CACHET: An interactive, incremental-attribution-based program transformation system for deriving incremental programs. In *Proceedings of the 10th Knowledge-Based Software Engineering Conference*, pages 19–26. IEEE CS Press, Los Alamitos, Calif., Nov. 1995.
27. Y. A. Liu. Principled strength reduction. In R. Bird and L. Meertens, editors, *Algorithmic Languages and Calculi*, pages 357–381. Chapman & Hall, London, U.K., 1997.

28. Y. A. Liu. Dependence analysis for recursive data. In *Proceedings of the IEEE 1998 International Conference on Computer Languages*, pages 206–215. IEEE CS Press, Los Alamitos, Calif., May 1998.
29. Y. A. Liu and G. Gómez. Automatic accurate time-bound analysis for high-level languages. In *Proceedings of the ACM SIGPLAN 1998 Workshop on Languages, Compilers, and Tools for Embedded Systems*, volume 1474 of *Lecture Notes in Computer Science*, pages 31–40. Springer-Verlag, June 1998.
30. Y. A. Liu and S. D. Stoller. Loop optimization for aggregate array computations. In *Proceedings of the IEEE 1998 International Conference on Computer Languages*, pages 262–271. IEEE CS Press, Los Alamitos, Calif., May 1998.
31. Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Discovering auxiliary information for incremental computation. In *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 157–170. ACM, New York, Jan. 1996.
32. Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Static caching for incremental computation. *ACM Trans. Program. Lang. Syst.*, 20(3):546–585, May 1998.
33. Y. A. Liu and T. Teitelbaum. Systematic derivation of incremental programs. *Sci. Comput. Program.*, 24(1):1–39, Feb. 1995.
34. D. Michie. "memo" functions and machine learning. *Nature*, 218:19–22, Apr. 1968.
35. D. J. Mostow and D. Cohen. Automating program speedup by deciding what to cache. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 165–172. Morgan Kaufmann Publishers, San Francisco, Calif., Aug. 1985.
36. R. Paige. Programming with invariants. *IEEE Software*, pages 56–69, Jan. 1986.
37. R. Paige. Symbolic finite differencing—Part I. In *Proceedings of the 3rd European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*, pages 36–56. Springer-Verlag, Berlin, May 1990.
38. R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Trans. Program. Lang. Syst.*, 4(3):402–454, July 1982.
39. H. A. Partsch. *Specification and Transformation of Programs—A Formal Approach to Software Development*. Springer-Verlag, Berlin, 1990.
40. A. Pettorossi. A powerful strategy for deriving efficient programs by transformation. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*. ACM, New York, Aug. 1984.
41. A. Pettorossi and M. Proietti. Rules and strategies for transforming functional and logic programs. *ACM Comput. Surv.*, 28(2):360–414, June 1996.
42. A. Pettorossi and M. Proietti. Program derivation via list introduction. In R. Bird and L. Meertens, editors, *Algorithmic Languages and Calculi*. Chapman & Hall, London, U.K., 1997.
43. W. Pugh. An improved cache replacement strategy for function caching. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, pages 269–276. ACM, New York, July 1988.
44. W. Pugh. The Omega Test: A fast and practical integer programming algorithm for dependence analysis. *Commun. ACM*, 31(8), Aug. 1992.
45. W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 315–328. ACM, New York, Jan. 1989.
46. P. W. Purdom and C. A. Brown. *The Analysis of Algorithms*. Holt, Rinehart and Winston, 1985.
47. T. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer-Verlag, New York, 1988.
48. W. L. Scherlis. Program improvement by internal specialization. In *Conference Record of the 8th Annual ACM Symposium on Principles of Programming Languages*, pages 41–49. ACM, New York, Jan. 1981.
49. D. R. Smith. KIDS: A semiautomatic program development system. *IEEE Trans. Softw. Eng.*, 16(9):1024–1043, Sept. 1990.
50. D. R. Smith. Structure and design of problem reduction generators. In B. Möller, editor, *Constructing Programs from Specifications*, pages 91–124. North-Holland, Amsterdam, 1991.
51. M. Sniedovich. *Dynamic Programming*. Marcel Dekker, New York, 1992.
52. B. Wegbreit. Goal-directed program transformation. *IEEE Trans. Softw. Eng.*, SE-2(2):69–80, June 1976.

Author Index

Abadi, M. 91

Barthe, G. 109
Benaissa, Z.E. 193
Benedikt, M. 2

Charatonik, W. 177

Danvy, O. 224

Erlich, Y-D. 258

Felleisen, M. 258
Flanagan, C. 91
Frade, M.J. 109

Gay, S. 74

Hill, P. 59
Hole, M. 74
Hudak, P. 1

Jansson, P. 273
Jeuring, J. 273

King, A 59
Krishnamurthi, S. 258

Liu, Y.A. 288

Moggi, E. 193
Müller, M. 177
Müller, P. 162
Mycroft, A. 208

Nielson, F. 20
Nielson, H.R. 20
Norrish, M. 147

Podelski, A. 177
Poetzsch-Heffter, A. 162

Reps, T. 2

Sagiv, M. 2
Sabelfeld, A. 40
Sands, D. 40
Shao, Z. 128
Sheard, T. 193
Smaus, J-G. 59
Stoller, S.D. 288

Taha, W. 193
Thiemann, P. 243
Trifonov, V. 128

Yang, Z. 224