

*Commenced Publication in 1973*

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

## Editorial Board

David Hutchison

*Lancaster University, UK*

Takeo Kanade

*Carnegie Mellon University, Pittsburgh, PA, USA*

Josef Kittler

*University of Surrey, Guildford, UK*

Jon M. Kleinberg

*Cornell University, Ithaca, NY, USA*

Alfred Kobsa

*University of California, Irvine, CA, USA*

Friedemann Mattern

*ETH Zurich, Switzerland*

John C. Mitchell

*Stanford University, CA, USA*

Moni Naor

*Weizmann Institute of Science, Rehovot, Israel*

Oscar Nierstrasz

*University of Bern, Switzerland*

C. Pandu Rangan

*Indian Institute of Technology, Madras, India*

Bernhard Steffen

*University of Dortmund, Germany*

Madhu Sudan

*Microsoft Research, Cambridge, MA, USA*

Demetri Terzopoulos

*University of California, Los Angeles, CA, USA*

Doug Tygar

*University of California, Berkeley, CA, USA*

Gerhard Weikum

*Max-Planck Institute of Computer Science, Saarbruecken, Germany*

Jens Palsberg (Ed.)

# Semantics and Algebraic Specification

Essays Dedicated to Peter D. Mosses  
on the Occasion of His 60th Birthday

Volume Editor

Jens Palsberg

University of California, Los Angeles

Department of Computer Science

4531K Boelter Hall, Los Angeles, CA 90095-1596, USA

E-mail: palsberg@ucla.edu

The cover illustration showing a view of Udine was retrieved from  
[http://commons.wikimedia.org/wiki/File:Old\\_udine.jpg](http://commons.wikimedia.org/wiki/File:Old_udine.jpg)

Library of Congress Control Number: 2009933274

CR Subject Classification (1998): F.3.2, F.3-4, D.2, D.1.5-6, I.2, I.1.3

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN 0302-9743

ISBN-10 3-642-04163-9 Springer Berlin Heidelberg New York

ISBN-13 978-3-642-04163-1 Springer 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. Violations are liable to prosecution under the German Copyright Law.

[springer.com](http://springer.com)

© Springer-Verlag Berlin Heidelberg 2009

Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India  
Printed on acid-free paper SPIN: 12750504 06/3180 5 4 3 2 1 0



**Peter D. Mosses**

# Preface

Peter Mosses, renowned researcher of Semantics of Programming Languages and Algebraic Specification Frameworks, turned 60 years old on November 3, 2008. To honor this event, many of Peter's coauthors, collaborators, close colleagues, and former students gathered in Udine, Italy on September 10, 2009 for a symposium in his honor. The presentations were on subjects related to Peter's many technical contributions and they were a tribute to his lasting impact on the field. Here is the program of the symposium:

- Opening: Jens Palsberg
- Session 1: (Chair: José Luiz Fiadeiro)
  - David Watt, Action Semantics in Retrospect
  - Hélène Kirchner, Component-Based Security Policy Design with Colored Petri Nets
  - José Meseguer, Order-Sorted Parameterization and Induction
- Session 2: (Chair: Andrzej Tarlecki)
  - Martin Musicante, An implementation of Object-Oriented Action Semantics in Maude
  - Christiano Braga, A Constructive Semantics for Basic Aspect Constructs
  - Bartek Klin, Structural Operational Semantics for Weighted Transition Systems
- Session 3:
  - Fernando Orejas, On the Specification and Verification of Model Transformations
  - Olivier Danvy, Towards Compatible and Interderivable Semantic Specifications for the Scheme Programming Language
  - Mark van den Brand, Type Checking Evolving Languages with MSOS
  - Edward Hermann Haeusler, Action Algebras and Model Algebras in Denotational Semantics
- Closing: Peter Mosses

Many thanks to Marina Lenisa from the University of Udine who coordinated the local arrangements. We also thank the Amga spa and the Net spa of Udine, the Municipality of Udine, the International Centre for Mechanical Sciences of Udine, and the Fondazione Crup for their financial support.

The 17 invited chapters of this Festschrift represent the proceedings of the symposium. Some contributors were unable to attend the event. The papers were reviewed by Philippe Bonnet, Doina Bucur, Will Clinger, Olivier Danvy, Edward Hermann Haeusler, Bartek Klin, Paddy Krishnan, Lars Kristensen, Søren Lassen, José Meseguer, Lasse R. Nielsen, Scott Owens, Jens Palsberg, Davide Sangiorgi, David Schmidt, Trian Serbanuta, Andrzej Tarlecki, and Claus Thrane. The reviewers provided feedback to the authors that helped them improve the papers.

Jill Edwards, Sara Fenn, Andy Gimblett, Will Harwood, Markus Roggenbach, and Monika Seisenberger provided a wealth of great photos of Peter. Special thanks to Olivier Danvy and José Luiz Fiadeiro for encouragement and help, and to Christopher Mosses, Peter's son, for helping to pick the photo of Peter that appears in this Festschrift.

September 2009

Jens Palsberg

# Table of Contents

Tribute to Peter Mosses .....	1
<i>Jens Palsberg</i>	
Action Semantics in Retrospect .....	4
<i>David A. Watt</i>	
Component-Based Security Policy Design with Colored Petri Nets .....	21
<i>Hejiao Huang and Hélène Kirchner</i>	
Order-Sorted Parameterization and Induction .....	43
<i>José Meseguer</i>	
An Implementation of Object-Oriented Action Semantics in Maude ....	81
<i>André Murbach Maidl, Cláudio Carvilhe, and Martin A. Musicante</i>	
A Constructive Semantics for Basic Aspect Constructs .....	106
<i>Christiano Braga</i>	
Structural Operational Semantics for Weighted Transition Systems ....	121
<i>Bartek Klin</i>	
On the Specification and Verification of Model Transformations .....	140
<i>Fernando Orejas and Martin Wirsing</i>	
Towards Compatible and Interderivable Semantic Specifications for the Scheme Programming Language, Part I: Denotational Semantics, Natural Semantics, and Abstract Machines .....	162
<i>Olivier Danvy</i>	
Towards Compatible and Interderivable Semantic Specifications for the Scheme Programming Language, Part II: Reduction Semantics and Abstract Machines .....	186
<i>Małgorzata Biernacka and Olivier Danvy</i>	
Type Checking Evolving Languages with MSOS .....	207
<i>M.G.J. van den Brand, A.P. van der Meer, and A. Serebrenik</i>	
Action Algebras and Model Algebras in Denotational Semantics .....	227
<i>Luiz Carlos Castro Guedes and Edward Hermann Haeusler</i>	
Mobile Processes and Termination .....	250
<i>Romain Demangeon, Daniel Hirschkoff, and Davide Sangiorgi</i>	
An Action Semantics Based on Two Combinators .....	274
<i>Kyung-Goo Doh and David A. Schmidt</i>	

X      Table of Contents

Converting between Combinatory Reduction Systems and Big Step Semantics .....	297
<i>Hanne Gottliebsen and Kristoffer H. Rose</i>	
Model-Based Testing and the UML Testing Profile .....	315
<i>Padmanabhan Krishnan and Percy Pari-Salas</i>	
A Complete, Co-inductive Syntactic Theory of Sequential Control and State .....	329
<i>Kristian Størvring and Soren B. Lassen</i>	
Vertical Object Layout and Compression for Fixed Heaps .....	376
<i>Ben L. Titzer and Jens Palsberg</i>	
<b>Author Index .....</b>	<b>409</b>

# Tribute to Peter Mosses

Jens Palsberg

UCLA Computer Science Department  
University of California, Los Angeles  
`palsberg@ucla.edu`

We offer this festschrift in tribute to Peter and in celebration of his 60th birthday.  
An academic symposium marked this event on September 10, 2009.

## Peter's Origins

Peter was born in 1948. He received in his BA and MSc degrees in Mathematics and his DPhil degree in Computer Science from University of Oxford in 1970, 1971, and 1975, respectively. His advisor was Christopher Strachey, one of the founding fathers of programming languages as we know them today. Indeed Peter was Strachey's last PhD student.

Peter went on to be a researcher at University of Oxford until 1976, then a faculty member at Aarhus University, Denmark from 1976 to 2004, including a period as head of the Department of Computer Science from 1995 to 1998, and then a professor at Swansea University, Wales from 2005, which is his current position. He has also held guest positions at University of Edinburgh, Scotland; SRI International, California; Stanford University, California; UFPE, Recife, Brazil; and Warsaw University, Poland.

## Peter's Research

Peter's many research interests span Semantics of Programming Languages and Algebraic Specification Frameworks. Peter has made fundamental contributions to Denotational Semantics, Operational Semantics, and Algebraic Semantics, and he has had lasting impact on how to combine Denotational Semantics and Algebraic Semantics, and on how to write more modular semantic descriptions. Peter wrote one of the best accounts of Denotational Semantics which appeared in the Handbook of Theoretical Computer Science in 1990, and he wrote the seminal book on Action Semantics in 1992. Peter is also the inventor of Abstract Semantics Algebras, Modular Structural Operational Semantics, and Unified Algebras, and a co-inventor of Algebraic Higher-Order Set Theory. Additionally, Peter was the founder and overall coordinator of CoFI, the Common Framework Initiative for algebraic specification and development of software, from 1995 to 1998. CoFI developed CASL, the Common Algebraic Specification Language, and Peter was the editor of the CASL Reference Manual and a coauthor of the CASL User Manual.

Peter's research has resulted in five software tools, namely the *Action Environment* for constructive action semantics, the *Prolog MSOS Tool* for teaching formal semantics using Modular Structural Operational Semantics, the *MAT* prototype tool for action semantics, the *ASD Tools* for supporting the original version of action semantics, and SIS, the first semantics implementation system for running programs according to their denotational semantics.

At last count, Peter has produced over 100 articles and papers. More than fifty of those papers are single-authored, a stunningly high number that bears witness to his scientific originality. Peter is also a great collaborator and has had forty-nine coauthors and coeditors so far:

- 1. Valentin M. Antimirov
- 2. Egidio Astesiano
- 3. Hubert Baumeister
- 4. Didier Bert
- 5. Michel Bidoit
- 6. Hendrik J. Boom
- 7. Christiano Braga
- 8. Mark van den Brand
- 9. Maura Cerioli
- 10. Christine Choppy
- 11. Arie van Deursen
- 12. Kyung-Goo Doh
- 13. Uffe H. Engberg
- 14. José Luiz Fiadeiro
- 15. Rob J. van Glabbeek
- 16. Carl A. Gunter
- 17. Edward Hermann Haeusler
- 18. Masami Hagiya
- 19. Anne Haxthausen
- 20. Claus Hintermeier
- 21. Takayasu Ito
- 22. Jørgen Iversen
- 23. Hélène Kirchner
- 24. Bernd Krieg-Brückner
- 25. Padmanabhan Krishnan
- 26. Gregory Kucherov
- 27. Kim G. Larsen
- 28. Søren Bøgh Lassen
- 29. Jan van Leeuwen
- 30. Pierre Lescanne
- 31. Andrew D. McGettrick
- 32. José Meseguer
- 33. Till Mossakowski
- 34. Hermano P. de Moura
- 35. Martín Musicante
- 36. Mark J. New
- 37. Claus B. Nielsen
- 38. Mogens Nielsen
- 39. Fernando Orejas
- 40. Gordon D. Plotkin
- 41. Charles Rattray
- 42. Donald Sannella
- 43. Michael I. Schwartzbach
- 44. Dana S. Scott
- 45. Andrzej Tarlecki
- 46. Robert D. Tennent
- 47. Irek Ulidowski
- 48. Osamu Watanabe
- 49. David A. Watt

In addition, Peter has friends and colleagues all over the world.

Peter has organized many international events, has been a steering committee member of four conference series, has been a chair or a member of many conference program committees, and has been a member of several advisory boards. He has also participated in IFIP WG 1.3 and 2.2, including as chair of IFIP WG 1.3 from 1998 to 2003.

## Peter's Students

Peter's students all recount his excellent and generous academic guidance, both technical and nontechnical. For one tiny example: Peter taught me to avoid abbreviations and implicit notation in formal descriptions; those may be good for the writer but they hinder the reader, and Peter's focus is unwaveringly on helping the reader.

Here is a list of Peter's doctoral descendants:

- Jens Palsberg (Aarhus University, 1992): *Provably Correct Compiler Generation*
  - Tian Zhao (Purdue University, 2002)
  - Dennis Brylow (Purdue University, 2003)
  - Di Ma (Purdue University, 2004)
  - Krishna Nandivada (UCLA, University of California, Los Angeles, 2005)
  - Christian Grothoff (UCLA, University of California, Los Angeles, 2006)
  - Benjamin Titzer (UCLA, University of California, Los Angeles, 2007)
  - Fernando Pereira (UCLA, University of California, Los Angeles, 2008)
- Claus Hintermeier (co-supervision, Université Henri Poincaré, Nancy, France, 1995): *Déduction avec Sortes Ordonnées et Égalités*
- Martín Musicante (co-supervision, Federal University of Pernambuco, Recife, Brazil, 1996): *On the Relational Semantics of Interleaving Constructors*
- Peter Ørbæk (Aarhus University, 1997): *Trust and Dependence Analysis*
- Søren Bøgh Lassen (Aarhus University, 1998): *Relational Reasoning about Functions and Nondeterminism*
- Christiano Braga (co-supervision, PUC Rio de Janeiro, Brazil, 2001): *Rewriting Logic as a Semantic Framework for Modular Structural Operational Semantics*
- Bartek Klin (Aarhus University, 2004): *An Abstract Coalgebraic Approach to Process Equivalence for Well-Behaved Operational Semantics*
- Jørgen Iversen (Aarhus University, 2005): *Formalisms and Tools Supporting Constructive Action Semantics.*

# Action Semantics in Retrospect

David A. Watt

Department of Computing Science, University of Glasgow,  
Glasgow G12 8QQ, Scotland  
`daw@dcs.gla.ac.uk`

**Abstract.** This paper is a themed account of the action semantics project, which Peter Mosses has led since the 1980s. It explains his motivations for developing action semantics, the inspirations behind its design, and the foundations of action semantics based on unified algebras. It goes on to outline some applications of action semantics to describe real programming languages, and some efforts to implement programming languages using action semantics directed compiler generation. It concludes by outlining more recent developments and reflecting on the success of the action semantics project.

## 1 Introduction

Action semantics arose out of Peter Mosses' dissatisfaction with existing semantic formalisms such as denotational semantics. He set out to develop a semantic formalism that would enable programming language descriptions to be comprehensible, modifiable, scalable, and reusable.

The action semantics project faced a number of challenges. It was necessary to design an action notation that would be powerful, natural, and yet manageable. Action semantics had to have solid foundations, in that the meaning of each action must be known precisely (and there must be no discrepancy between theory and intuition). Action semantics had to be tested in practice by writing descriptions of a variety of real programming languages.

All these challenges were addressed vigorously. Peter designed an action notation with a readable English-like syntax. He established its foundations by developing and using a new algebraic formalism called unified algebras. The operational flavour of action notation made it very natural for describing the dynamic semantics of programming languages (without forcing authors or readers to master the details of an abstract machine, as other operational formalisms do.) This same operational flavour also made action semantics directed compiler generation an attractive prospect.

A small but active action semantics community grew up during the 1980s and 1990s. To enable this community to keep in contact and exchange ideas, Peter organized a series of action semantics workshops [18,23,21].

The rest of this paper is structured as follows. Section 2 explains the dissatisfaction with denotational semantics (and other semantic formalisms) that motivated the development of action semantics. Section 3 recounts the early ideas

and the eventual design of action semantics. Section 4 outlines the foundations of action semantics, in terms of the algebraic properties of action combinators, the new framework of unified algebras, and the operational semantics of action notation. Section 5 recounts some of the attempts to apply action semantics to the static and dynamic semantics of real programming languages. Section 6 briefly discusses some of the compiler-generation projects based on action semantics. Section 7 summarizes some more recent developments. Section 8 concludes by reflecting on the extent to which the action semantics project succeeded in its original aims. The appendices contain illustrative action semantic descriptions of three small programming languages, each built on its predecessor.

## 2 Motivations

As a research student at Oxford University in the 1970s, Peter gained a deep understanding of denotational semantics. He wrote a denotational description of Algol-60 [10]. He also developed his Semantic Implementation System (SIS) [11] which, given a denotational description of a programming language, would generate a compiler for that language. Later he wrote an overview of denotational semantics [16].

Experience shows that denotational descriptions have many pragmatic problems. Peter clearly perceived that the fundamental problem is the use of the lambda calculus as a notation for defining the semantic functions and auxiliary functions. Fundamental concepts such as binding, storing, sequencing, and choice must be *encoded* in the lambda calculus, rather than being expressed directly. Any change in the structure of a semantic function forces changes to all the semantic equations that define or use that semantic function.

To illustrate these points, suppose that we are developing a programming language incrementally. We will start with a simple applicative language APP; then add assignments to make a simple imperative language IMP; and finally add exceptions to make a language EXC.

If we are using denotational semantics to describe each successive language, the development might proceed as follows. When we describe the applicative language APP, the denotation of an expression is a function that maps an environment to a value. When we extend to the imperative language IMP, in which an expression may have side-effects, the denotation of an expression must map an environment and a store to a value and an updated store. When we extend to the language EXC, in which an expression might throw an exception, further wholesale changes are necessary, perhaps moving to the continuation style. In summary, the semantic functions for expressions change as follows:

- (APP)  $\text{eval } \_ : \text{Expression} \rightarrow (\text{Env} \rightarrow \text{Value})$
- (IMP)  $\text{eval } \_ : \text{Expression} \rightarrow (\text{Env} \rightarrow \text{Store} \rightarrow \text{Value} \times \text{Store})$
- (EXC)  $\text{eval } \_ : \text{Expression} \rightarrow (\text{Env} \rightarrow (\text{Value} \rightarrow \text{Cont}) \rightarrow \text{Cont})$

Even quite modest language extensions can force structural changes to the semantic functions, and hence changes to all the semantic equations defining or

using these semantic functions. If the language were further extended to support concurrency, further wholesale changes would be needed, now employing the heavy machinery of power-domains.

A denotational description of a small programming language is feasible, but a denotational description of a realistic language on the scale of Pascal or Java constitutes a formidable challenge. In practice, nearly every published denotational description has been incomplete, typically omitting whatever the author deemed to be non-core features of the language.

Even experts find denotational descriptions hard to understand. To non-experts they appear unfamiliar, unnatural, and incomprehensible. For this reason, denotational descriptions have almost never been used in language reference manuals.

These pragmatic problems are not unique to denotational descriptions; in fact, they are shared by all the other well-known semantic formalisms. For example, in natural semantics [8], the designer who is developing a programming language incrementally must be prepared to change the structure of the judgements:

- (APP)  $e \vdash E \Rightarrow v$
- (IMP)  $e, s \vdash E \Rightarrow v, s'$
- (EXC)  $e, s \vdash E \Rightarrow v/x, s'$

(where  $E$  is an expression,  $e$  is an environment,  $v$  is a value, and  $v/x$  is either a value or an exception). Each change in a judgement forces changes to all semantic rules using that judgement.

So Peter set himself the challenge of devising a semantic formalism that would enable semantic descriptions to have good pragmatic properties. More precisely, semantic descriptions should be:

- comprehensible, i.e., able to be understood (at least at an intuitive level) by non-experts;
- modifiable, i.e., able to adapt to changes in the design of the described language, without disproportionate effort;
- scalable, i.e., able to grow proportionately to the size of the described language;
- reusable, i.e., able to be used (at least in part) in the description of a related language.

### 3 Inspirations

Peter's original idea (possibly inspired by Backus [1]) was to describe the semantics of a programming language's constructs using a set of abstract semantic entities, which he called *actions* [13,14]. Each action represents a computation that receives and produces information. Simple actions represent elementary computations (such as binding an identifier to a value or storing a value in a cell). Compound actions can be composed from simple actions by means of *combinators*. Each combinator captures a standard form of control flow (sequencing,

interleaving, or choice). Each combinator also captures a standard form of information flow (distribution, composition, etc.).

Thus Peter had already established the basis of what he later christened *action semantics*. A number of details still remained to be resolved. In particular, what set of simple actions and what set of combinators would be adequate to describe a variety of language constructs? Would a small number of simple actions and a small number of combinators be sufficient? What notation should be used to write down these simple actions and combinators? And would the resultant semantic descriptions truly be comprehensible, modifiable, scalable, and reusable?

Peter originally employed a very concise but cryptic notation for the actions. Later he replaced this with a more readable notation; for example,  $\nabla$  became complete; ' $A_1 \oplus A_2$ ' became ' $A_1$  or  $A_2$ ', ' $A_1 \otimes A_2$ ' became ' $A_1$  and  $A_2$ ', and ' $A_1 \odot A_2$ ' became ' $A_1$  then  $A_2$ '. Careful choice of names for primitives and combinators made it possible to write quite complex actions in an English-like notation that is comprehensible even to non-experts.

An important aspect of Peter's original idea was to factor the actions into *facets*. Each simple action has an effect in only one facet. A compound action may have an effect in more than one facet. Any action may be polymorphically extended to other (possibly unforeseen) facets, in which its behaviour is neutral. The significance of this is that it enables action terms (and hence semantic descriptions) to be modifiable and reusable.

An action can *complete*, *escape*, *fail*, or *diverge*.

The *basic facet* is concerned only with control flow. Exclusive choice, whereby only one of the sub-actions is performed, is exemplified by the combinator ' $_1$  or  $_2$ '. Sequencing, whereby the second sub-action is performed only if and when the first sub-action has completed, is exemplified by the combinator ' $_1$  and then  $_2$ '. Interleaving, whereby the sub-actions may be performed in any order, is exemplified by the combinator ' $_1$  and  $_2$ '. Every combinator adopts one of these three patterns of control flow.

In the *functional facet* an action receives and produces *transient* data. The simple action 'give  $d$ ' produces the datum yielded by  $d$ . The simple action 'check  $b$ ', completes if  $b$  yields true but fails otherwise. These and some other simple actions contain terms called *yielders*, which allow these actions to use received information. The yielder 'the given  $s$ ' yields the received transient datum (which must be of sort  $s$ ). For example, 'give successor of the given integer' uses a received integer and produces a different integer.

In the *declarative facet* an action receives and produces *bindings*, which are associations between *tokens* and *bindable* data. The simple action 'bind  $k$  to  $d$ ' produces a binding of the token  $k$  to the datum yielded by  $d$ . The yielder 'the  $s$  bound to  $k$ ' yields the datum (of sort  $s$ ) bound to the token  $k$ .

When an action is polymorphically extended to another facet, it produces no information in that other facet. For example, 'give 7' produces a single transient but no bindings; 'bind "n" to 3' produces a single binding but no transients; 'give the integer bound to "n"' uses a received binding and produces a transient. A

compound action may produce both transients and bindings, for example, ‘give 7 and bind “n” to 3’.

The idea that actions are polymorphically extensible to other facets enables a reasonably small set of combinators to describe the control flow and information flow of a variety of language constructs. The number of possible combinators is constrained by the fact that the information flow must be consistent with the control flow. In particular, an interleaving combinator cannot make transients or bindings flow from one sub-action to the other; a sequential combinator cannot make transients or bindings flow from the second sub-action to the first sub-action.

The combinator ‘\_ and \_’, distributes received information and combines produced information. In ‘ $A_1$  and  $A_2$ ’, received transients and bindings are distributed to both  $A_1$  and  $A_2$ , transients produced by  $A_1$  and  $A_2$  are combined by tupling, and bindings produced by  $A_1$  and  $A_2$  are combined by disjoint union. For example, if the compound action ‘give successor of the given integer and bind “m” to the given integer’ receives the transient value 5, both sub-actions will receive that 5, and the compound action will produce the transient 6 and a binding of “m” to 5.

The combinator ‘\_ then \_’ behaves as functional composition in the functional facet. In ‘ $A_1$  then  $A_2$ ’, received transients are passed into  $A_1$ , transients produced by  $A_1$  are passed into  $A_2$ , and transients produced by  $A_2$  are produced by the compound action. In the declarative facet, ‘\_ then \_’ distributes received bindings and combines produced bindings. For example, if the compound action ‘give successor of the given integer then bind “m” to the given integer’ receives the transient value 5, it will produce a binding of “m” to 6 (but will produce no transients).

Conversely, the combinator ‘\_ hence \_’ behaves as functional composition in the declarative facet, but in the functional facet it distributes received transients and combines produced transients. This combinator captures the concept of scope: the bindings produced by  $A_1$  are used in  $A_2$ , and nowhere else.

Remarkably, the handful of combinators already mentioned, plus a few others, turn out to be adequate to describe the vast majority of language constructs. There is very little need for more specialized combinators.

In the *imperative facet*, actions allocate, inspect, and update the store. The store is structured as a mapping from *cells* to *storable* data. Stored information is stable: updates cannot be forgotten, but can be superseded by later updates. The simple action ‘allocate a cell’ produces a previously-unallocated cell. The simple action ‘store  $d$  in  $c$ ’ updates the cell yielded by  $c$  to contain the datum yielded by  $d$ . The yielder ‘the  $s$  stored in  $c$ ’ inspects the cell yielded by  $c$ , which must contain a datum of sort  $s$ . Commands in imperative languages can be described very naturally by imperative actions.

In the *communicative facet*, a number of *agents* can be created, each charged with performing a particular action with its own local store. Agents send and receive messages to one another asynchronously. Communicated information is

permanent: a message once sent can never be retrieved nor superseded. Communicative actions can be used to describe concurrent languages.

There are no combinators specifically associated with the imperative and communicative facets. Each combinator's behaviour in the basic facet controls the order in which imperative and communicative sub-actions are performed. In particular, ‘`_ and _`’ and similar combinators allow compound imperative and concurrent sub-actions to be interleaved in a nondeterministic fashion.

## The Applicative Language (APP)

Appendix A shows an action semantic description (ASD) of the applicative language APP.

Two semantic functions are introduced here. The semantic function ‘evaluate  $\_$ ’ maps each expression to an action that will produce a single value. The semantic function ‘elaborate  $\_$ ’ maps each declaration to an action that will produce bindings.

We see here the notation used for sorts of actions. The sort ‘action’ includes all actions. The subsort ‘action [giving a value]’ includes only those actions that produce a single value – this is the sort of ‘evaluate  $E$ ’. The subsort ‘action [binding]’ includes only those actions that produce bindings – this is the sort of ‘elaborate  $D$ ’.

The compound action ‘furthermore  $A$ ’ overlays the received bindings by the bindings produced by  $A$ .

Functions in APP are modeled by *abstractions*, each of which encapsulates an action. The operation ‘abstraction of  $A$ ’ creates an abstraction encapsulating the action  $A$ . The operation ‘closure of  $a$ ’ injects the received bindings into the abstraction yielded by  $a$  (such that these bindings will be received by the encapsulated action when it is eventually performed). Similarly, the operation ‘application of  $a$  to  $d$ ’ injects the datum yielded by  $d$  into the abstraction  $a$ . Finally, the simple action ‘enact  $a$ ’ performs the action encapsulated by the abstraction  $a$ . The notation for subsorts of abstractions parallels that for subsorts of actions, so functions are modeled by abstractions of subsort ‘abstraction [using the given value | giving a value]’; this means that each abstraction encapsulates an action that both receives a value and produces a value.

## The Imperative Language (IMP)

We can easily extend our applicative language to an imperative language IMP, whose ASD is shown in Appendix B.

Each expression is now mapped to an action of sort ‘action [giving a value | storing]’, which includes those actions that both produce a value and (potentially) update the store. This reflects the fact that expressions now have side-effects.

Each declaration is now mapped to an action of sort ‘action [binding — storing]’, which includes only those actions that produce bindings and update the store. This reflects the fact that declarations now have side-effects (in particular, variable declarations create and initialize cells). Semantic equations have been added for the new constructs: assignment expressions and variable declarations.

Despite the changes to the denotations, very little change is needed to the existing semantic equations. The semantic equation for ‘evaluate  $I$ ’ must be modified to take into account of the fact that in IMP the identifier  $I$  could be bound to a cell. None of the other semantic equations need be changed. For example, in ‘evaluate  $\llbracket E_1 + E_2 \rrbracket$ ’, the sub-expressions  $E_1$  and  $E_2$  are meant to be evaluated collaterally, and for this the combinator ‘ $\_$  and  $\_$ ’ is still appropriate. (On the other hand, if we decided that the sub-expressions should be evaluated sequentially, we would simply replace the ‘ $\_$  and  $\_$ ’ combinator by the ‘ $\_$  and then  $\_$ ’ combinator.)

### The Imperative Language with Exceptions (EXC)

We can further extend the language to allow expressions and declarations to throw exceptions, leading to the language EXC whose ASD is shown in Appendix C.

Each expression is now mapped to an action of sort ‘action [giving a value | storing | escaping with an exception]’, which includes only actions that either complete giving a value or escape giving an exception, updating the store in either case. The denotations of declarations have been changed likewise. Despite these changes to the denotations, *none* of the existing semantic equations needs to be changed. Semantic equations have been added for the new constructs: expressions that throw and catch exceptions.

The simple action ‘escape with  $d$ ’ escapes producing the datum yielded by  $d$  as a transient. When an action escapes, enclosing actions escape too, except where the ‘ $\_$  trap  $\_$ ’ combinator is used. The action ‘ $A_1$  trap  $A_2$ ’ will perform  $A_2$  if and only if  $A_1$  escapes, in which case any transients produced by  $A_1$  are received by  $A_2$ .

The combinators used in the ASD of IMP are still appropriate in the ASD of EXC. For example, the semantic equation for ‘evaluate  $\llbracket E_1 + E_2 \rrbracket$ ’ still works because the ‘ $\_$  and  $\_$ ’ combinator causes the compound action to escape if either sub-action escapes.

## 4 Foundations

The pragmatic benefits of action semantics would amount to little if the action notation were not well-founded. Peter addressed this problem with his usual energy and his deep understanding of the foundations of computation. He tackled the problem at three levels.

### Algebraic Properties

The action combinators have a number of simple algebraic properties, unsurprisingly. For example, the combinator ‘ $\_$  or  $\_$ ’ is total, associative, and commutative, and has a simple action (fail) as its unit; similarly, the combinators ‘ $\_$  and  $\_$ ’, ‘ $\_$  and then  $\_$ ’, ‘ $\_$  then  $\_$ ’, and ‘ $\_$  hence  $\_$ ’, are all total and associative, and each has a simple action as its unit.

These algebraic properties can be used to reason about compound actions. They are insufficient to define the meanings of these actions precisely. However, they are sufficient to reduce the whole of action notation to a moderately-sized kernel. For full details see Appendix B of Peter's book [17].

## Unified Algebras

Peter invented a wholly new formalism of *unified algebras*. As its name suggests, this formalism is characterized by a unified treatment of ordinary values (*individuals*) and sorts (*choices*). Each individual is just a singleton sort. Each operation maps a choice to a choice. Unified algebras are in fact an independent contribution to formal methods, and applicable well beyond action semantics.

The clause:

- $\text{truth-value} = \text{true} \mid \text{false}$  (*individual*) .

defines **truth-value** to be the choice between the individuals **true** and **false**. The further clauses:

- $\text{not } \_ : \text{truth-value} \rightarrow \text{truth-value}$  (*total*) .
- $\text{not false} = \text{true} ; \text{not true} = \text{false}$  .

introduce the usual '**not**  $\_$ ' operation.

A more interesting example is:

- $0 : \text{natural}$  .
- $\text{successor of } \_ : \text{natural} \rightarrow \text{natural}$  (*total*) .
- $\text{natural} = 0 \mid \text{successor of natural}$  .

which defines **natural** to be the choice among the individuals **0**, **successor of 0**, **successor of successor of 0**, etc. The operation '**successor of**  $\_$ ' maps any choice of natural numbers to the choice of their successors; in particular, '**successor of natural**' maps the natural numbers to the positive integers.

In this way Peter defined the *data notation* that underlies action notation. Data notation includes partial and total orders, tuples, truth-values, numbers, characters, lists, strings, syntax, sets, and maps. For full details see Appendix E of [17].

The author of an ASD of a particular programming language also uses unified algebras to define sorts (such as **bindable** and **storable**) that are used but not defined by action notation; and to define sorts specific to the described language.

## Semantics of Action Notation

Peter also used unified algebras to define *action notation* itself. We can now see that **action** is the choice among all actions, and that **action[...]** is a restriction of that choice. We have already seen examples such as '**action [giving a value]**', '**action [binding]**', and '**action [binding | storing]**'.

The meanings of actions (and yielders) in the kernel of action notation are defined using structural operational semantics (SOS) [26]. This handles all facets,

including the communicative facet. It is a ‘small-step’ semantics, and correctly defines interleaving and nondeterminism. A step in a sequencing action such as ‘ $A_1$  and then  $A_2$ ’ is a step in  $A_1$  (unless  $A_1$  has terminated). A step in an interleaving action such as ‘ $A_1$  and  $A_2$ ’ can be a step in either  $A_1$  or  $A_2$  (unless one of these sub-actions has terminated). A step in an exclusive choice action such as ‘ $A_1$  or  $A_2$ ’ can be a step in either  $A_1$  or  $A_2$  (unless one of these sub-actions has committed – i.e., has performed an irreversible step such as updating the store or sending a message – in which case the other sub-action is abandoned).

The SOS of action notation is expressed in the notation of unified algebras. Although unified algebra operations are functions, their results can be *choices*. This allows nondeterminism to be expressed. Consider, for example, the operation that performs a single step in the action ‘ $A_1$  and  $A_2$ ’; its result is the choice of configurations that can arise from performing *either* a single step in  $A_1$  *or* a single step in  $A_2$ .

For full details see Appendix C of [17].

## 5 Applications

To test the pragmatic properties of action semantics, it was necessary to write ASDs of real programming languages. The first such test was an ASD of Pascal developed by Peter and me [22]. This was followed by ASDs of various imperative, object-oriented, functional, and concurrent programming languages, including CCS and CSP [4], Java [3], and Standard ML [28]. In his book [17], Peter shows the incremental development of an ASD of a substantial subset of Ada, including tasks.

These tests convincingly demonstrated that action semantics does indeed have the desired pragmatic properties. It is feasible to build complete ASDs of real programming languages. It is possible to reuse parts of these ASDs to describe related languages, even where the successor language is significantly richer than its predecessor. (For example, I reused large parts of the Pascal ASD in an ASD for Modula-3, which extends Pascal with objects, exceptions, and concurrency.) The ASDs are comprehensible and (in the usual sense of the word) natural – certainly far more so than corresponding denotational or natural semantic descriptions would be.

Although action notation was originally designed to describe dynamic semantics, Peter and I found that it can also be used to describe static semantics. This idea was tested on Pascal [22] and Standard ML [28]. Again, these tests demonstrated that action semantics is usable. In the Pascal static semantics, the enforcement of nominal type equivalence required the creation of a new type at each type-denoter, and this was encoded (rather unnaturally, it must be admitted) using the imperative facet. The Pascal static semantics thus reads like a type-checking algorithm. Likewise, in the first version of my ML static semantics, an auxiliary action ‘unify  $_1$  with  $_2$ ’ encoded the unification algorithm. A later version of my ML static semantics used unified algebra notation rather than action notation. The semantic function for each expression yields a *choice* of types

- the expression’s principal type and all its instances. This gave the ML static semantics a more natural relational flavour.

## 6 Implementations

As we have seen, Peter started his academic career by building a compiler generator, SIS, based on denotational semantics [11]. Later he addressed the problem of compiler correctness [12].

Action notation has an operational flavour (unlike the lambda calculus). It directly supports the basic computational concepts of ordinary programming languages (rather than encoding them in terms of mathematical abstractions). So it is natural to think of generating compilers from action semantics.

An ASD of programming language  $L$  can be seen as specifying a translation from the abstract syntax of  $L$  to action notation. If a parser for  $L$  is composed with the  $L$ -to-action-notation translator and an interpreter for action notation, the product is an interpretive compiler for  $L$ . If a parser for  $L$  is composed with the  $L$ -to-action-notation translator and a code generator for action notation, the product is a full compiler for  $L$ .

Several action-notation interpreters have been written. My first interpreter, written in ML, covered the functional, declarative, and imperative facets of the action notation kernel, but did not support nondeterminism, interleaving, or the communicative facet. Hermano Moura’s interpreter, also written in ML, was more general. Later I wrote an action-notation interpreter in ASF+SDF, which did support nondeterminism.

In action notation the imperative facet is well-behaved, since store updates are stable. (This differs from denotational semantics, where stores are variables like any other, so stores can be cloned and updates can be reversed.) Overall, action notation behaves sufficiently like conventional imperative code to make it feasible to translate action notation to reasonably efficient object code.

*Actress* [2] was a compiler generator based on a large subset of action notation (excluding mainly the communicative facet). Given the ASD of any programming language  $L$ , Actress could generate a compiler composed of:

- a parser, which was generated from  $L$ ’s syntax, and which translated the source code to an abstract syntax tree (AST);
- an AST-to-action-notation translator, which was generated from  $L$ ’s semantics;
- an action-notation sort-checker, which inferred the sorts of data in the functional and declarative facets;
- an action-notation transformer, which performed various transformations of the action notation, including algebraic simplifications, bindings elimination, and stack storage allocation;
- an action-notation code generator, which translated action notation to object code expressed in C.

Actress could generate a better-quality compiler if  $L$  was statically-scoped and statically-typed, but it did not insist on these properties.

*Cantor* [25] was a compiler generator based on a broadly similar subset of action notation. Given the ASD of a statically-scoped and statically-typed language  $L$ , Cantor could generate a compiler composed of:

- a parser, which was generated from  $L$ 's syntax, and which translated the source code to an AST;
- an AST-to-action-notation translator, which was generated from  $L$ 's semantics;
- an action-notation code generator, which translated action notation to idealized SPARC assembly code;
- an assembler, which translated the idealized SPARC assembly code to real machine code.

Significantly, Cantor came with a correctness proof, which verified *inter alia* that the generated compiler would translate statically-typed source code to object code that will not misbehave despite being untyped.

*Oasis* [24] was a compiler generator that accepted an ASD expressed in Scheme syntax. Given the ASD of a statically-scoped and statically-typed language  $L$ , Oasis could generate a compiler composed of:

- a parser, separately generated from  $L$ 's syntax (using YACC, say);
- an AST-to-action-notation translator, which translated the AST into Scheme syntax, combined it with the ASD already expressed in Scheme, and then interpreted the resulting Scheme program to generate action notation;
- an action-notation code generator, which translated action notation to SPARC assembly code;
- an assembler, which translated the SPARC assembly code to real machine code.

Oasis's code generator employed a variety of forward and backward analyses, which *inter alia* distinguished between bindings of known and unknown values, achieved constant propagation, and discovered opportunities for stack storage allocation. Oasis-generated compilers could generate remarkably efficient object code.

Because action notation directly reflects the fundamental concepts of programming languages, it seems to be feasible for a compiler generator automatically to discover key properties of the described language  $L$ , and to exploit these properties to improve the generated compiler and/or its object code. Here are some examples of such properties:

- Does  $L$  require stack allocation and/or heap allocation?
- Is  $L$  statically-scoped or dynamically-scoped?
- Is  $L$  statically-typed or dynamically-typed?

It is quite easy for a compiler generator to determine whether  $L$  is statically-scoped or dynamically-scoped by analyzing the semantics of  $L$ . If bindings are always injected into an abstraction (by means of the ‘closure of  $\lambda$ ’ operation) when the abstraction is created,  $L$  is statically-scoped. If bindings are always injected into an abstraction when the abstraction is enacted,  $L$  is dynamically-scoped.

It is rather more difficult for a compiler generator to determine whether the described language  $L$  is statically-typed or dynamically-typed. If both a static semantics and a dynamic semantics of  $L$  are provided, these would have to be analyzed together. A simpler approach is to focus on  $L$ 's dynamic semantics, which invariably contains implicit sort information. (For example, consider the semantic equations for ‘evaluate  $\llbracket E_1 = E_2 \rrbracket$ ’ and ‘evaluate  $\llbracket \text{“if” } E_1:\text{Expression} \text{ “then” } E_2:\text{Expression} \text{ “else” } E_3:\text{Expression} \rrbracket$ ’ in Appendix A.) Analysis of the ASD opens up the prospect of automatically extracting type rules and determining whether  $L$  is statically-typed. The simplest approach of all is sort inference on a program-by-program basis, which is essential if the generated compiler is to generate efficient object code. The richness and generality of unified algebras make all these approaches challenging, but they have been extensively studied, most notably by David Schmidt's group [6,7,27].

## 7 Later Developments

### Modular Structural Operational Semantics

SOS suffers from the same pragmatic problems as other semantic formalisms: any change in the structure of the configurations forces a major rewrite. The SOS of action notation was not immune to these problems.

Peter, with his characteristic energy, developed what he called *modular structural operational semantics* (MSOS) [19]. MSOS abstracts away from the structure of the configurations in somewhat the same manner as action semantics abstracts away from the structure of denotations. Peter then used MSOS to rewrite the semantics of action notation [20].

### Action Notation Revisited

When the design of action notation was complete, it turned out to be larger than anticipated. In particular, there were fifteen action combinators in the kernel action notation (although six of these were rarely used). Moreover, there was an overlap between yielders and functional actions: both receive information and produce data. There was duplication between actions and abstractions: the action and abstraction subsort notations were similar, and for each action combinator there was a corresponding (but rarely-used) abstraction combinator. The communicative facet was complicated, and yet made it difficult to express the semantics of threads.

Working with Søren Lassen, Peter took a fresh look at action notation. They removed yielders from the kernel, but kept them in the full notation for the sake of fluency. They allowed actions to be treated as data, thereby eliminating the separate notion of an abstraction (and its associated notation). They also removed a number of features that were rarely-used or of doubtful utility. They completely redesigned the communicative facet, most significantly allowing agents to share a global store.

The resulting proposed version of action notation, *AN-2* [9], turned out to be about half the size of its predecessor, and its kernel was slightly smaller too.

## Reusing Programming Language Descriptions

One of the main aims of action semantics was reuse of programming language descriptions, particularly when describing a new language that inherits features from an older language. This has been demonstrated informally, as we saw in Section 5.

Working with Kyung-Goo Doh, Peter set out to demonstrate this more formally [5]. Their idea was to build a library of common programming language constructs, in which each module essentially contains a single semantic equation. For example, we could have a module for while-commands, a module for procedure calls with call-by-value semantics, a module for procedure calls with call-by-name semantics, and so on. Building an ASD of a new programming language then consists largely of selecting the appropriate modules from the library. Any conflicts that might arise (for example, if both call-by-value and call-by-name semantics are selected and they are not syntactically distinguished) can be detected readily.

## 8 Conclusion

Has the action semantics project been successful?

In theoretical terms, the answer is unequivocally yes. Action semantics has proved to be powerful enough to describe a large variety of programming language constructs. Action-semantic descriptions never rely on semi-formal notational conventions (unlike most denotational and natural semantic descriptions). Action notation has a secure foundation, and the drive to provide a secure foundation has produced remarkable spin-offs in unified algebras and MSOS. Another spin-off has been that the facet structure of action notation provides an intellectual framework for understanding the fundamental concepts of programming languages [29].

In practical terms, the answer is more equivocal. Action semantics has made it possible, for the first time, to write semantic descriptions of real programming languages that are comprehensible (even to non-experts), modifiable, scalable, and reusable. Action semantics directed compiler generation has been demonstrated to be feasible. However, engineering a high-quality compiler generator is a long-term project, and the necessary sustained effort has not yet been achieved. The action semantics community grew rapidly and spread over five continents, but it never exceeded 20 researchers at any one time, not enough to secure a long-term critical mass of activity.

I would like to take this opportunity to acknowledge the many contributions of the action semantics community, and above all to acknowledge the energetic leadership of Peter Mosses. This paper has set out to be an impressionistic overview rather than a comprehensive account of the action semantics project. Almost certainly it does not do justice to everyone's contribution, for which I can only apologize.

## References

1. Backus, J.: Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM* 21, 613–641 (1978)
2. Brown, D.F., de Moura, H.P., Watt, D.A.: Actress: an action semantics directed compiler generator. In: Kastens, U., Pfahler, P. (eds.) CC 1992. LNCS, vol. 641, pp. 95–109. Springer, Heidelberg (1992)
3. Brown, D.F., Watt, D.A.: JAS: a Java action semantics. In: [23], pp. 43–56
4. Christensen, S., Olsen, M.H.: Action semantics of CCS and CSP, Report DAIMI IR-44, Computer Science Department, Aarhus University (1988)
5. Doh, K.-G., Mosses, P.D.: Composing programming languages by combining action-semantics modules, BRICS Report Series RS-03-53, Computer Science Department, Aarhus University (2003)
6. Doh, K.-G., Schmidt, D.A.: Extraction of strong typing laws from action semantics definitions. In: Krieg-Brückner, B. (ed.) ESOP 1992. LNCS, vol. 582, pp. 151–166. Springer, Heidelberg (1992)
7. Even, S., Schmidt, D.A.: Type inference for action semantics. In: Jones, N.D. (ed.) ESOP 1990. LNCS, vol. 432, pp. 71–95. Springer, Heidelberg (1990)
8. Kahn, G.: Natural semantics. In: Brandenburg, F.J., Wirsing, M., Vidal-Naquet, G. (eds.) STACS 1987. LNCS, vol. 247, pp. 22–39. Springer, Heidelberg (1987)
9. Lassen, S., Mosses, P.D., Watt, D.A.: an introduction to AN-2: the proposed new version of action notation. In: [21], pp. 19–36
10. Mosses, P.D.: The mathematical semantics of Algol60, Technical Monograph PRG-12, Programming Research Group, Oxford University (1974)
11. Mosses, P.D.: Mathematical semantics and compiler generation, DPhil thesis, Oxford University (1975)
12. Mosses, P.D.: A constructive approach to compiler correctness. In: de Bakker, J.W., van Leeuwen, J. (eds.) ICALP 1980. LNCS, vol. 85. Springer, Heidelberg (1980)
13. Mosses, P.D.: Abstract semantic algebras! In: Bjørner, D. (ed.) Formal Description of Programming Concepts II, North-Holland, Amsterdam (1983)
14. Mosses, P.D.: A basic abstract semantic algebra. In: Plotkin, G., MacQueen, D.B., Kahn, G. (eds.) Semantics of Data Types 1984. LNCS, vol. 173. Springer, Heidelberg (1984)
15. Mosses, P.D.: Unified algebras and action semantics. In: Cori, R., Monien, B. (eds.) STACS 1989. LNCS, vol. 349, pp. 17–35. Springer, Heidelberg (1989)
16. Mosses, P.D.: Denotational semantics. In: van Leeuwen, J., et al. (eds.) Handbook of Theoretical Computer Science, pp. 575–631. Elsevier, Amsterdam (1990)
17. Mosses, P.D.: Action Semantics. Cambridge Tracts in Theoretical Computer Science (1992)
18. Mosses, P.D. (ed.): First International Workshop on Action Semantics, BRICS Notes Series NS-94-1, Computer Science Department, Aarhus University (1994)
19. Mosses, P.D.: Foundations of modular structural operational semantics. In: Kutyłowski, M., Wierzbicki, T., Pacholski, L. (eds.) MFCS 1999. LNCS, vol. 1672. Springer, Heidelberg (1999)
20. Mosses, P.D.: A modular SOS for action notation. In: [23], pp. 131–142
21. Mosses, P.D., de Moura, H.P. (eds.): Third International Workshop on Action Semantics, BRICS Notes Series NS-00-6, Computer Science Department, Aarhus University (2000)

22. Mosses, P.D., Watt, D.A.: Pascal: action semantics, version 0.6 (unpublished) (1993)
23. Mosses, P.D., Watt, D.A. (eds.): Second International Workshop on Action Semantics, BRICS Notes Series NS-99-3, Computer Science Department, Aarhus University (1999)
24. Ørbæk, P.: Oasis: an optimizing action-based compiler generator. In: Fritzson, P.A. (ed.) CC 1994. LNCS, vol. 786, pp. 1–15. Springer, Heidelberg (1994)
25. Palsberg, J.: A provably correct compiler generator. In: Krieg-Brückner, B. (ed.) ESOP 1992. LNCS, vol. 582, pp. 418–434. Springer, Heidelberg (1992)
26. Plotkin, G.D.: A structural approach to operational semantics, Report DAIMI FN-19, Computer Science Department, Aarhus University (1981)
27. Schmidt, D.A., Doh, K.-G.: The facets of action semantics - some principles and applications. In: [18], pp. 1–15
28. Watt, D.A.: The static and dynamic semantics of Standard ML. In: [23], pp. 155–172
29. Watt, D.A.: Programming Language Design Concepts. Wiley, Chichester (2004)

## A ASD of the Applicative Language (APP)

### Semantic Entities

- bindable = value | function .
- value = truth-value | integer .
- function = abstraction [using the given value | giving a value] .

### Semantics of Expressions

- evaluate \_ : Expression → action [giving a value] .
- (1) evaluate  $\llbracket L:\text{Literal} \rrbracket$  =
    - give the literal value of  $L$  .
  - (2) evaluate  $\llbracket I:\text{Identifier} \rrbracket$  =
    - give the value bound to  $I$  .
  - (3) evaluate  $\llbracket E_1 "+" E_2 \rrbracket$  =
    - | evaluate  $E_1$  and evaluate  $E_2$
    - then give sum of (the given integer#1, the given integer#2) .
  - (4) evaluate  $\llbracket E_1 "=" E_2 \rrbracket$  =
    - | evaluate  $E_1$  and evaluate  $E_2$
    - then give (the given value#1 is the given value#2) .
  - (5) evaluate  $\llbracket I "(" E ")" \rrbracket$  =
    - evaluate  $E$  then
    - enact application of (the function bound to  $I$ ) to the given value .
  - (6) evaluate  $\llbracket \text{"let"} D:\text{Declaration} \text{ "in"} E:\text{Expression} \rrbracket$  =
    - furthermore elaborate  $D$
    - hence evaluate  $E$  .
  - (7) evaluate  $\llbracket \text{"if"} E_1:\text{Expression} \text{ "then"} E_2:\text{Expression} \text{ "else"} E_3:\text{Expression} \rrbracket$  =
    - evaluate  $E_1$  then
      - | check the given value is true then evaluate  $E_2$
      - or
      - | check the given value is false then evaluate  $E_3$  .

## Semantics of Declarations

- elaborate \_ : Declaration → action [binding] .
- (1) elaborate  $\llbracket \text{"val" } I:\text{Identifier} \text{ "}" E:\text{Expression} \rrbracket =$   
evaluate  $E$  then bind  $I$  to the given value .
  - (2) elaborate  $\llbracket \text{"fun" } I_1:\text{Identifier} \text{ "(" } I_2:\text{Identifier} \text{ ")" "}" E:\text{Expression} \rrbracket =$   
bind  $I_1$  to closure of abstraction of  
    - | furthermore bind  $I_2$  to the given value
    - | hence evaluate  $E$  .

## B ASD of the Imperative Language (IMP)

### Semantic Entities

- bindable = value | function | cell .
- storable = value .
- value = ...
- function = abstraction [using the given value | giving a value | storing] .

### Semantics of Expressions

- evaluate \_ : Expression → action [giving a value | storing] .
- (1) evaluate  $\llbracket L:\text{Literal} \rrbracket = \dots$
  - (2) evaluate  $\llbracket I:\text{Identifier} \rrbracket =$   
give the value bound to  $I$  or  
give the value stored in the cell bound to  $I$  .
  - (3) evaluate  $\llbracket E_1 "+" E_2 \rrbracket = \dots$
  - (4) evaluate  $\llbracket E_1 "=" E_2 \rrbracket = \dots$
  - (5) evaluate  $\llbracket I "(" E ")" \rrbracket = \dots$
  - (6) evaluate  $\llbracket \text{"let" } D:\text{Declaration} \text{ "in" } E:\text{Expression} \rrbracket = \dots$
  - (7) evaluate  $\llbracket \text{"if" } E_1:\text{Expression} \text{ "then" } E_2:\text{Expression} \text{ "else" } E_3:\text{Expression} \rrbracket =$   
...
  - (8) evaluate  $\llbracket I:\text{Identifier} \text{ ":" } E:\text{Expression} \rrbracket =$   
evaluate  $E$  then  
    - | store the given value in the cell bound to  $I$  and
    - | give the given value .

### Semantics of Declarations

- elaborate \_ : Declaration → action [binding | storing] .
- (1) elaborate  $\llbracket \text{"val" } I:\text{Identifier} \text{ "}" E:\text{Expression} \rrbracket = \dots$
  - (2) elaborate  $\llbracket \text{"fun" } I_1:\text{Identifier} \text{ "(" } I_2:\text{Identifier} \text{ ")" "}" E:\text{Expression} \rrbracket = \dots$
  - (3) elaborate  $\llbracket \text{"var" } I:\text{Identifier} \text{ ":" } E:\text{Expression} \rrbracket =$   
    - | evaluate  $E$  and allocate a cell
    - then
    - | store the given value#1 in the given cell#2 and
    - | bind  $I$  to the given cell#2 .

## C ASD of the Imperative Language with Exceptions (EXC)

### Semantic Entities

- bindable = value | function | cell | exception .
- storable = ...
- value = ...
- function = abstraction [using the given value | giving a value | storing | escaping with an exception] .
- exception  $\leq$  distinct-datum .

### Semantics of Expressions

- evaluate \_ : Expression → action [giving a value | storing | escaping with an exception] .
- (1) evaluate  $\llbracket L:\text{Literal} \rrbracket = \dots$
  - (2) evaluate  $\llbracket I:\text{Identifier} \rrbracket = \dots$
  - (3) evaluate  $\llbracket E_1 "+" E_2 \rrbracket = \dots$
  - (4) evaluate  $\llbracket E_1 "=" E_2 \rrbracket = \dots$
  - (5) evaluate  $\llbracket I "(" E ")" \rrbracket = \dots$
  - (6) evaluate  $\llbracket \text{"let"} D:\text{Declaration} \text{"in"} E:\text{Expression} \rrbracket = \dots$
  - (7) evaluate  $\llbracket \text{"if"} E_1:\text{Expression} \text{"then"} E_2:\text{Expression} \text{"else"} E_3:\text{Expression} \rrbracket = \dots$
  - (8) evaluate  $\llbracket I:\text{Identifier} ":"= E:\text{Expression} \rrbracket = \dots$
  - (9) evaluate  $\llbracket \text{"throw"} I:\text{Identifier} \rrbracket =$   
    escape with the exception bound to  $I$  .
  - (10) evaluate  $\llbracket E_1 \text{"catch"} I:\text{Identifier} \text{"then"} E_2 \rrbracket =$   
        evaluate  $E_1$   
        trap  
             $\left| \begin{array}{l} \text{check the given exception is the exception bound to } I \\ \text{then evaluate } E_2 \end{array} \right.$   
            or  
             $\left| \begin{array}{l} \text{check not the given exception is the exception bound to } I \\ \text{and then escape with the given exception} . \end{array} \right.$

### Semantics of Declarations

- elaborate \_ : Declaration → action [binding | storing | escaping with an exception] .
- (1) elaborate  $\llbracket \text{"val"} I:\text{Identifier} "=" E:\text{Expression} \rrbracket = \dots$
  - (2) elaborate  $\llbracket \text{"fun"} I_1:\text{Identifier} "(" I_2:\text{Identifier} ")" "=" E:\text{Expression} \rrbracket = \dots$
  - (3) elaborate  $\llbracket \text{"var"} I:\text{Identifier} ":"= E:\text{Expression} \rrbracket = \dots$

# Component-Based Security Policy Design with Colored Petri Nets<sup>☆</sup>

Hejiao Huang<sup>1,2</sup> and Hélène Kirchner<sup>2</sup>

<sup>1</sup> Harbin Institute of Technology Shenzhen Graduate School, China

<sup>2</sup> INRIA Bordeaux Sud-Ouest, France

**Abstract.** Security policies are one of the most fundamental elements of computer security. This paper uses colored Petri net process (CPNP) to specify and verify security policies in a modular way. It defines fundamental policy properties, i.e., completeness, termination, consistency and confluence, in Petri net terminology and gets some theoretical results. According to XACML combiners and property-preserving Petri net process algebra (PPPA), several policy composition operators are specified and property-preserving results are stated for the policy correctness verification.

**Keywords:** security policy, colored Petri net, specification and verification, property-preservation.

*This paper is dedicated to Peter D. Mosses, whose many scientific contributions, from the design of CASL [1] to his vision of programming languages description [43], have advocated a component-based approach improving reusability and the importance of tool support.*

## 1 Introduction

A security policy is a definition of what it means to be secure for a system, an organization or another entity. For an organization, it addresses the constraints on behavior of its members as well as constraints imposed on adversaries by mechanisms such as doors, locks, keys and walls. For information systems, the security policy addresses constraints on functions and flow among them, constraints on access by external systems and adversaries including programs and access to data by people. In computer security, a rigorous and formal approach in the design and implementation of policies is crucial in order to ensure security of transactions and of access to resources. Today the security policies have to be deployed in the actual context of distributed organizations, mobile users and on-line services, which requires them to be easily composable, highly reconfigurable, time dependent and reactive to their environment.

In large systems, there are many classes of subjects with different needs for processing a variety of resources. Different subjects usually have different (even competing) requirements on the use of resources and their security goals (confidentiality, availability,

---

\* This work was supported in part by National Natural Science Foundation of China with Grant No. 10701030.

integrity) may be distinct. Hence, various access requirements have to be consistently authorized and maintained. Detecting and solving conflicts is crucial for the policy designers who need analysis and verification tools, as well as methodology for conflict resolution. Suitable properties of policies, such as consistency, have to be clearly identified and proved. Formal methods and logical approaches provide already some help but should be better tuned to the specific domain and properties of security policies. This is especially important for the formalization and understanding of specific properties such as privacy or trust.

Security is seldom identified with a single, simple policy; the policies and resources of a system may be modeled, built or owned by different unrelated parties, at different times, and under different environments. A lot of research is currently developed on security and privacy for pervasive computing. The interested reader can refer to [15] for a nice introduction to these problems. A rational way to organize the multiple cooperating components and sub-policies is needed to achieve a complete secure system. In this setting, the theory of security policy composition becomes crucially significant. The idea is similar to the component-based design in software engineering: each simple and original module is first specified independently, then based on the control flow of the system or policy requirement, the modules are composed together into a whole system model. The objective is to deduce the properties of the whole system, based on the properties of the sub-modules, according to theoretical results about property-preservation. In our context, the global policy is considered as loosely-coupled sub-policies and should preserve the properties of the constituent sub-policies. To build a complex global policy, it is yet a challenge to provide various composition operators ensuring by constructing the safe integration of component policies.

Several approaches have been followed for the formal specification, verification and modular design of security policies. Some of them are reviewed in the next Section 2. Since 2005, a formal way of specifying policy with strategic rewriting and verifying the policy properties expressed in this setting has been developed. The properties of confluence, consistency, termination and completeness have emerged as useful and verifiable properties of rewrite-based policies. Based on this experience and previous work, the idea is to explore with a similar approach another formalism, the Petri net model.

So this paper addresses the specification of security policies in the Petri net formalism and their component-based architecture borrowed from the software engineering approach. It provides the following contributions:

1. It defines Colored Petri Net Process (CPNP) that enrich colored Petri nets with interfaces needed to specify requests and decisions of security policies. It gives formal definitions for policy-related properties, namely, completeness, termination, consistency and confluence. These properties, presented in [23] in the rewriting framework, are here adapted to the Petri net approach and given in Petri net terminology. Some theoretical results concerning these properties are stated.
2. It specifies some policy composition operators based on Petri nets. This paper specifies four composition combiners, provided in the eXtensible Access Control Markup Language (XACML), a declarative access control policy language. These so-called XACML combiners are dedicated to those policies with decision conflicts which will be solved according to predefined rules. They are easily expressed

as CPNP. Then, relying on property-preserving Petri net process algebra (PPPA), a technology in Petri net theory applicable mainly for component-based system design in software engineering, three simple composition operators are specified in order to give some hints on applying PPPA to the security policy design. The composition operators in PPPA are mainly useful for building policies in a modular way. For each composition operator, the preservation of policy properties is studied.

The remaining part of the paper is organized as follows: after a short survey of some related works in Section 2, Section 3 gives basic terminology about colored Petri nets and the formal definition of policy properties with Petri net terminology. Related results concerning the policy properties are also given in this section. Section 4 is about the component-based security policy design technology. Classical XACML combiners are specified with CPNP, and other composition operators are defined. For each composition operator, property-preservation results are presented. At last, some conclusive remarks are given in Section 5.

## 2 Related Works

An important part of security is brought by access control whose aim is to prevent resources from unauthorized accesses. An access control policy is a set of rules specifying that a given user is allowed (or denied) to perform an action on a given object. Different access control models have been proposed to specify the authorization mechanism: discretionary [26] and mandatory [8,4] access control models, role-based access control (RBAC) [47], organization-based access control (OrBAC) [34], Chinese-Wall policy [12]... Such models are useful to analyze whether some security property (such as confidentiality or separation of duties) is satisfied in all possible states of the system.

Even in more general contexts than access control, policies are often expressed by rules in natural language: if some conditions are satisfied, then, grant or deny some request. Not surprisingly, their specification and verification can rely on various logic-based formalisms. The existing approaches have adopted different semantic frameworks such as first-order logic [25,29,24,31], Datalog [11,39,10,22], temporal logic [5,19], or deontic logic [16,34]. In open and distributed property-based access control systems, tasks are delegated and agents must rely on others to perform actions that they themselves cannot do. The concept of trust has been formalized and models of trust together with logical formalisms have been proposed, such as for instance Wooldridge's Logic Of Rational Agents [30]. These formalisms provide different degrees of expressiveness for authoring a policy.

The conflict resolution problem that may appear in particular when composing policies has motivated a lot of research. The idea of disambiguating among possibly conflicting decisions appear in several works, such as in [29,34]. It is also at the core of the industrial standard access-control language XACML [42] that provides several combiners to solve such conflicts, developed later on in this paper.

Policy composition is addressed in [9] through an algebra of composition operators that is able to define union, intersection, policy templates, among other operations. The work presented in [52] extended this algebra with negative authorizations and non-determinism and also includes an operator for sequential composition. An alternative

for composing access control policies in Java applications is implemented by the Polymer system [3]. An approach based on defeasible logic and non monotonic inference is taken in [38] for composing policy specifications for web-services security. In [13,14], the author proposes a set of high-level composition operators coherent with a four-valued logic for policies.

More recently, term rewriting theory has been applied for the specification and verification of security policy design and their composition [23,7,6,45]. In a rewrite-based specification, policies are expressed by rules close to natural language: if some conditions are satisfied, then, a given request is evaluated into a decision, for instance it may be granted or denied. This expressivity allows one to finely specify the conditions under which decision takes place and these conditions may involve attributes related to subjects or resources. Moreover strategic rewriting is used to express control on the rules and to handle priorities or choices between possible decisions. For instance the specification of XACML combiners is given in [23,45] in the rewriting context. The rewrite-based approach provides executable specifications for security policies, which can be independently designed, verified, and then anchored on programs [20] using a modular discipline. In [18], the rewriting approach has been used to explore the information flow over the reachable states of a system and detect information leakage. In [35], a policy analysis method is proposed. It is based on strategic narrowing, that provides both the necessary abstraction for simulating executions of the policy over access requests and the mechanism for solving *what-if* queries from the security administrator.

This formal specification approach was also influenced by the Common Framework Initiative for algebraic specification and development (CoFI) that produces the Common Algebraic Specification Language CASL [1], dedicated to the formal specification of functional requirements and modular design of software. An area of particular interest for applications is that of reactive, concurrent, distributed and real-time systems. Extensions of CASL to deal with systems of this kind, and methods for developing software from such specifications have been explored for a different kind of formalism provided by Petri nets. An example is given in [17] where this CASL methodology for specification development is used to provide systematic guidelines for Petri nets specifications.

Petri nets are well known for their graphical and analytical capabilities for the specification and verification of concurrent and distributed systems. Using Petri nets for the specification and verification of the security policy design is not a new story. In [46], a colored Petri net (CPN) based framework is presented for verifying the consistency of RBAC policies. The reachability analysis technique is applied for RBAC policy verification. In [41], CPN is used to specify a real industrial example, namely an access control system developed by the Danish security company Dalcotech A/S. Based on the CPN model, the Design/CPN tool is applied for the implementation of automatic code generation. [36] defines task based access control as a dynamic workflow and then specifies the workflow with Petri nets. [53] and [54] model Chinese wall policy and Strict Integrity Policy, respectively, with CPN and apply coverability graph for the verification. [33] uses CPN for the specification of mandatory access control policies and occurrence graph is applied for verification. [21] applies Predicate/Transition net for the modeling and analysis of software security system architectures.

The common characteristic of these technologies is that for a specific security policy, Petri nets are used for the specification, and the reachability-tree related techniques and CPN Tools are applied for the verification. However it is well-known that these techniques will face the state explosion problem when the system is large and complex. In order to overcome this shortage and to strengthen the advantages of the Petri net formalism both in security policy and in software engineering areas, it seems reasonable to apply CPN for the security policy design specification and verification in a modular way.

### 3 Colored Petri Net Process and Policy Properties

In this paper, we assume some familiarity with basics notions on Petri nets, briefly introduced here. More terminology and fundamentals on Petri net theory can be found in [44,27].

#### 3.1 Colored Petri Nets

**Definition 1.** (*Petri nets, PN*) A Petri net  $(N, M_0)$  is a net  $N = (P, T, F, W)$  with an initial marking  $M_0$  where,

- $P$  is a finite set of places of cardinality  $|P|$ ;
- $T$  is a finite set of transitions such that  $P \cap T = \emptyset$ , and  $P \cup T \neq \emptyset$  ;
- $F \subseteq (P \times T) \cup (T \times P)$  is the flow relation;
- $W$  is a weight function such that  $W(x, y) \in N^+$  if  $(x, y) \in F$  and  $W(x, y) = 0$  if  $(x, y) \notin F$ ;
- $M_0$  is a function  $M : P \rightarrow N$  such that  $M(p)$  represents the number of tokens in place  $p \in P$ .

**Definition 2.** (*pre-set, post-set, input set and output set*) For  $x \in P \cup T$ ,  $\bullet x = \{y | (y, x) \in F\}$  and  $x^\bullet = \{y | (x, y) \in F\}$  are called the pre-set (input set) and post-set (output set) of  $x$ , respectively. For a set  $X \subseteq P \cup T$ ,  $\bullet X = \cup_{x \in X} \bullet x$  and  $X^\bullet = \cup_{x \in X} x^\bullet$ .

**Definition 3.** (*Colored Petri nets, CPN*) A colored Petri net  $CPN = (N, M_0, C)$  is a Petri net with a color set  $C$  assigned to the tokens in the places and a weight assigned to each arc.

This definition of colored Petri net is much simpler than the standard definition that can be found in [32], but it is quite enough for our specification purpose.

Usually, colors in the CPN are used to distinguish different types of data. In a CPN, the marking and the weight are denoted as multiple dimensional vectors. A transition  $t \in T$  is firable (or enabled) at a marking  $M$  if and only if  $\forall p \in P : M(p) \geq W(p, t)$ , where both  $M(p)$  and  $W(p, t)$  are  $|C|$ -dimensional vectors, compared component wise. Firing (or executing) transition  $t$  results in changing marking  $M$  to marking  $M'$ , where  $\forall p \in P : (M'(p) = M(p) - W(p, t) + W(t, p))$ . In what follows, for better readability, we often identify the function  $W$  with its image, i.e. the set  $W = \{W(p, t) | (p, t) \in F\}$ .

**Definition 4.** (*Firing sequence and reachability*) Let  $M, M'$  be markings,  $t$  be a transition, and  $\sigma$  be a transition sequence (called firing sequence) in a Petri net  $(N, M_0)$ .  $M[N, \sigma]M'$  means that  $M'$  is reachable from  $M$  by firing  $\sigma$ .  $R(N, M)$  denotes the reachability set of  $N$  starting from  $M$ , i.e., the smallest set of markings such that: (a)  $M \in R(N, M)$ ; (b) If  $M' \in R(N, M)$  and  $M'[N, t]M''$  for some  $t \in T$ , then  $M'' \in R(N, M)$ .

**Definition 5.** (*Colored Petri Net Process, CPNP*) (Fig. 1)

A Colored Petri Net Process (CPNP)  $B = (CPN, p_e, p_x)$  is a colored Petri net CPN with an additional unique entry place  $p_e$  and unique exit place  $p_x$ , where the place  $p_e$  (resp.  $p_x$ ) has no input (resp. output) transitions.

For denoting a marking of CPNP, we have different expressions that may be used interchangeably throughout the paper:

- a vector expression: e.g., for a CPNP with two colors  $c_1, c_2$ , a marking with three places may be  $M = ((2, 1), (0, 0), (1, 1))$ , meaning that there are two tokens with color  $c_1$ , one token with color  $c_2$  in place  $p_1$ , no tokens in place  $p_2$  and two tokens with color  $c_1$  and  $c_2$  respectively in place  $p_3$ ;
- a multiset expression: e.g., for the above marking, we denote  $M = \{2c_1, c_2\}p_1 + \{c_1, c_2\}p_3$ . Correspondingly, sometimes we use just a color set to denote the marking in a place; e.g.,  $M(p_1) = \{2c_1, c_2\}$ , and the number of tokens in place  $p_1$  is  $|M(p_1)| = 3$ ;
- the mix of vector and multi-set expression. e.g., for the above marking  $M$  and a new place  $p_4$  with one token, we may use  $M + p_4$  to denote another marking which includes places  $p_1, p_2, p_3$  and  $p_4$ .

When the security policy is specified with a CPNP, the entry place  $p_e$  represents the request, while the exit place  $p_x$  represents the decision (see Fig 1). Different requests and different decisions are distinguished with different colored tokens. At the same time, the entry place and exit place are designed as two interfaces of the process. The initial marking of a CPNP is denoted as  $M_e = M_0 + p_e$ , the exit marking is  $M_x = M + p_x$ , where  $M$  is a marking in the internal CPN.

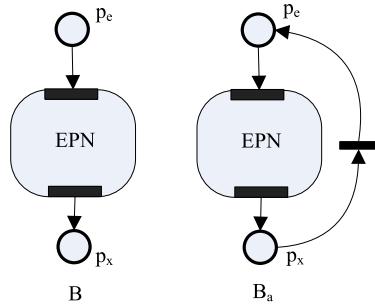
Since a CPNP is not strongly connected, it cannot satisfy those important system properties such as liveness, reversibility and so on. However, they can be recovered by considering the associated net of a CPNP  $B$ , that is a net with an additional transition  $t$  and two arcs  $(p_x, t)$  and  $(t, p_e)$  in  $B$  (Fig. 1). A CPNP is called almost live (respectively, bound, reversible, etc) if its associated net is live (respectively, bound, reversible, etc).

In this paper, modular security policy will be considered and the basic modules will be specified with CPNP. Correspondingly, the policy properties will be defined on CPNP.

### 3.2 Completeness

A security policy is decision complete (or simply complete) if it computes at least one decision for every incoming request. This property is also called totality in [2] and [50].

**Definition 6.** (*Completeness*) Let a security policy be specified with a CPNP  $B = (CPN, p_e, p_x)$ . The policy  $B$  is complete if for any initial marking  $M_e$ , there exists a marking  $M_x = M + p_x$  which is reachable from  $M_e$ .



**Fig. 1.** A CPNP and its associated net

Based on the definition, an initial marking represents a request, and the exit marking  $M + p_x$  is reachable, implying that the policy will return a decision.

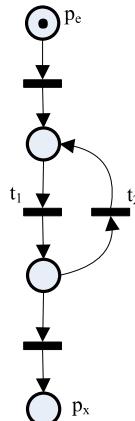
### 3.3 Termination

A security policy terminates if the evaluation of every incoming request terminates.

**Definition 7.** (*Termination*) Let a security policy be specified with a CPNP  $B = (\text{CPN}, p_e, p_x)$ . The policy  $B$  is terminating (or strongly terminating) iff  $B$  has no infinite firing sequences.  $B$  is said weakly terminating iff  $B$  has at least one finite firing sequence. If  $B$  terminates with an exit marking  $M_x = M_0 + p_x$ ,  $B$  is called properly terminating.

If  $B$  is weakly terminating, it may have infinite firing sequence(s) but must terminate in some cases (see Fig 2). Strong termination requires that the policy always terminates with a finite number of firing steps; while properly terminating requires the policy to terminate (strongly or weakly) and to reach a special exit state.

The following results allow us to connect these policy properties and the usual notions of boundedness, reversibility, liveness and deadlock-freeness in Petri nets [44].



**Fig. 2.** An example of a weakly terminating policy

**Proposition 1.** Let a security policy be specified with a CPNP  $B = (CPN, p_e, p_x)$ . The following properties hold:

1. If  $B$  is almost live,  $B$  is complete;
2. If  $B$  terminates properly,  $B$  is complete;
3. If  $B$  terminates from any reachable marking and deadlock-free,  $B$  is complete;
4. If  $B$  is complete and almost bounded, then it terminates (strongly or weakly).
5. If  $B$  is almost live and almost bounded, then it is terminating.
6. If  $B$  is almost live and almost reversible, then it is properly terminating.

*Proof.* 1. If  $B$  is almost live, then for the initial marking  $M_e$  specifying a request, there exists a reachable marking such that the transition  $t \in \bullet p_x$  is firable and  $M_x = M + p_x$  is reached after firing the transition  $t$ .

2. If  $B$  terminates properly, then  $M_x = M_0 + p_x$  is a reachable marking and hence  $B$  is complete.

3. By contradiction. If  $B$  is not complete, then there exists a reachable marking such that either  $B$  cannot terminate or it reaches a dead marking. This is in contradiction with the assumption.

4. By contradiction. If  $B$  cannot terminate, then either there is a cycle or  $B$  has an infinite firing sequence. As a result, either the tokens keep on transferring in the cycle and  $B$  cannot be complete or the associated net of  $B$  is unbounded. This is in contradiction with the assumption.

5. If  $B$  is almost live, then  $B$  is complete (based on 1), and by property 4,  $B$  is terminating.

6. Since  $B$  is almost live, by property 1,  $B$  is complete. That is,  $M_x = M + p_x$  is reachable from  $M_e = M_0 + p_e$ . Since  $B$  is almost reversible,  $M_0 + p_x$  is reachable from  $M + p_x$  in the associated net  $B_a$ . Hence  $M_x = M + p_x = M_0 + p_x$  in  $B$  and  $B$  is properly terminating.

### 3.4 Consistency

A security policy is consistent if it computes at most one access decision for any given input request.

**Definition 8. (Consistency)** Let a security policy be specified with a CPNP  $B = (CPN, p_e, p_x)$ . Then the policy  $B$  is consistent iff for any request  $M_e = M_0 + p_e$ , all the reachable markings  $M_i$  satisfy that  $|M_i(p_x)| \leq 1$  and for any exit marking  $M_j$  and  $M_k$ ,  $M_j(p_x) = M_k(p_x)$ .

Consistency implies that for any request, the policy will return at most one decision. According to the above definition, all reachable markings can have at most one token ( $|M_i(p_x)| \leq 1$ ) with a unique identical color in place  $p_x$  (since  $M_j(p_x) = M_k(p_x)$ ). In the case where the CPNP does not terminate, the decision place  $p_x$  will not be marked.

The consistency property can be related to state equations in Petri net theory. In a general Petri net  $(N, M_0)$ , any reachable marking  $M$  satisfies the state equation  $M = M_0 + V\mu_i$ , where  $V$  is the incidence matrix and  $\mu_i$  is the count vector of a firing sequence  $\sigma$  and  $M_0[N, \sigma]M$  [44].

**Proposition 2.** *Let a security policy be specified with a CPNP  $B$ . Then,  $B$  is consistent if*

- at most one of the following state equations is satisfied:

$$M_i = M_e + V\mu_i, \text{ where } |M_i(p_x)| = 1, i = 1, 2 \dots$$

- and none of the following state equations is satisfied:

$$M_i = M_e + V\mu_i, \text{ where, } |M_i(p_x)| > 1, i = 1, 2 \dots$$

*Proof.* If none of the first state equations is satisfied, then the policy cannot make a decision. If at most one of first state equations is satisfied, this implies that there may exist a decision.

If the second equation cannot be satisfied, this implies that the case of more than two different decisions is impossible.

Let us now relate consistency and the notion of confluence defined for Petri nets.

### 3.5 Confluence

**Definition 9.** (*Confluence*) *Let a security policy be specified with a CPNP  $B = (CPN, p_e, p_x)$ . The policy  $B$  is confluent iff for any initial marking  $M_e = M_0 + p_e$  and any two reachable markings  $M_i, M_j \in R(B, M_e)$ , there exists a reachable marking  $M_c$  in  $B$  such that  $M_c \in R(B, M_i) \cap R(B, M_j)$ .*

If there exists a set of markings, denoted as  $HS$ , reachable from  $M_i, M_j \in R(B, M_e)$ ,  $HS$  is called the home space of  $B$ . When  $HS$  has only one element  $M_c$ ,  $M_c$  is called the home marking of  $B$ .

The confluence property has been studied in the literature [49,51,37]. It is proved to be a decidable property in Petri net theory. For ordinary Petri nets (with weight 1 on each arc and no self-loop), the confluence can be reduced to confluence of a 2-shallow term rewriting systems [37]. The following Proposition is extracted from [37].

**Proposition 3.** 1. *If a Petri net has a home marking then it is confluent.* 2. *A safe Petri net (i.e., a PN which satisfies that the number of tokens in any place cannot exceed one for any reachable marking) has a home marking iff it is confluent.* 3. *Any confluent and strongly terminating Petri net has a unique home marking.*

In the context of colored Petri net process, we get the following result.

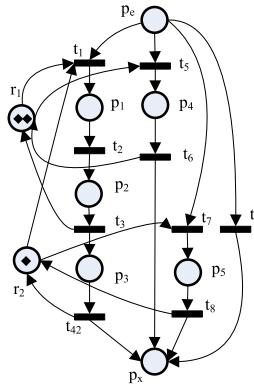
**Proposition 4.** *Let a security policy be specified with a CPNP  $B$ . If  $B$  is consistent and properly terminating, then  $B$  is confluent and has a unique home marking.*

*Proof.* Since  $B$  is properly terminating and consistent, for any request marking  $M_e = M_0 + p_e$  there exists a unique exit marking  $M_x = M_0 + p_x$  reachable from  $M_e$ . For any two reachable markings  $M_i$  and  $M_j$ , they can reach  $M_x$ . Hence  $B$  is confluent and  $M_x$  is the unique home marking.

### 3.6 An Example

This example illustrates how to apply the concepts and results introduced above to policy verification.

Fig. 3 is a CPNP based specification of a resource accessing policy. There are two printers (resource place  $r_1$ ) and one copying machine (place  $r_2$ ) for accessing. The policy is that once the resource is available, the access request is permitted. Based on the CPNP specification, a user can successfully request printing (firing  $t_5$ ), or copying (firing  $t_7$ ), or printing and copying (firing  $t_1$ ), or doing nothing (firing  $t_9$ ) once the requested resource is available.



**Fig. 3.** An example for CPNP-based policy verification

It is easy to verify that the CPNP model is almost live, bounded and almost reversible (by observation or by CPN-Tool). Based on Proposition 1, it is complete and properly terminating. Furthermore, for any request (e.g., printing) there exists a unique firing sequence (e.g.,  $t_5t_6$  for the request of “printing”) satisfying the state equation. By Proposition 2, it is consistent. By Proposition 4, it is confluent and has a unique home marking  $M_x = 2r_1 + r_2 + p_x$ .

## 4 Component-Based Policy Composition Based on CPNP

In this section, our focus is on the composition of the security policies in a modular way. In general, combining security policies may result in inconsistent or non-terminating policies. In a first part, we concentrate on the classical XACML policy combiners, whose purpose is to solve conflicts that may occur when two policies are combined. They are formalized in the colored Petri net approach. In a second part, three useful composition operators are presented to build policies in a modular way. We explore the preservation by these operators of the properties of completeness, consistency, termination and confluence of the component policies.

#### 4.1 CPNP-Based Specification of Policy Composition According to XACML Combiners

For a security system, in particular an access control system, the same resource may be governed by different policies and their respective decisions may be different. XACML policy combiners are used to solve the conflicts resulting from applying different policies for the same resources. There are four combiners described as follows:

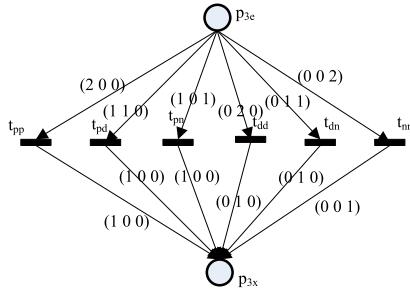
- *Permit-overrides*: whenever one of the sub-policies answers to a request with a “Permit” decision, the final authorization for the composed policy is “Permit”. The policy will generate a “Deny” only in the case where at least one of the sub-policies returns “Deny”, and all others return “NotApplicable” or “Indeterminate”. When all sub-policies return “NotApplicable”, the final output is “NotApplicable”. The decision is “Indeterminate” if no sub-policy returns a decision, i.e. when unexpected errors occur in every evaluation attempt.
- *Deny-overrides*: the semantics is similar to *Permit-overrides*. The only difference is to exchange “Permit” and “Deny” in the above description.
- *First-applicable*: the final authorization coincides with the result of the first sub-policy which produces the decision “Permit” or “Deny”; if no sub-policy is applicable, then the decision is “NotApplicable”, if mistakes occur, then it is “Indeterminate”.
- *Only-one-applicable*: the resulting decision will be “Permit” or “Deny” if the single sub-policy that applies to the request generates one of these decisions. The result will be “NotApplicable” if all sub-policies return such decision. The result is “Indeterminate” if more than one policy returns a decision different from “NotApplicable”.

In order to simplify the specification model, we assume in this paper that there are only two sub-policies and no mistake occurs in the combiners, so there are only three possible decisions, namely “Permit”, “Deny”, and “NotApplicable”. Actually it would not be harder but only more technical to handle multiple sub-policies and an additional decision “Indeterminate”.

Let us first consider the formal specification of Permit-overrides combiner based on colored Petri net processes. The Petri net structure given in Figure 4 is defined as  $POC = (p_{3e}, p_{3x}, T_c, F_c, W_c, M_c, C_c)$ . Places  $p_{3e}$  and  $p_{3x}$  have three types of tokens colored with “Permit”, “Deny”, “NotApplicable” respectively, representing the three different decisions. Weights are assigned to each arc. For example, in Figure 4, the weight of the arc  $(p_{3e}, t_{pp})$  is  $(2, 0, 0)$ : this means that firing transition  $t_{pp}$  requires at least two tokens colored with “Permit”, i.e., both sub-policies return the decision “Permit”. After firing transition  $t_{pp}$ , the output is  $(1, 0, 0)$ , meaning that place  $p_{3x}$  gets a token colored with “Permit”, i.e., the final decision is “Permit”.

For the Deny-overrides combiner (DOC), the specification and the composition technique are similar to the above defined Permit-overrides combiner, just exchanging “Permit” and “Deny”.

The First-applicable combiner  $FAC = (\{p_{4e}, p_{4x}\}, T_c, F_c, W_c, M_c, C_c)$  is defined in Figure 5, where the entry place  $p_{4e}$  represents all possible decisions taken by the sub-policies, while the exit place  $p_{4x}$  represents the final decision of the composed policy.  $T_c$

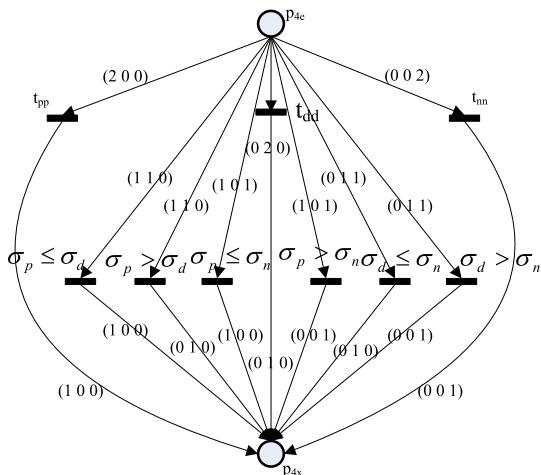


**Fig. 4.** Permit-overrides combiner POC

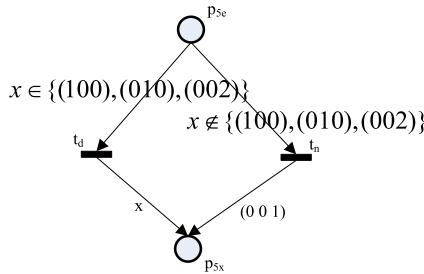
is the transition set: for each transition, there is a firing condition assigned to it. For example, the weight  $(1, 0, 1)$  means that the decisions of the sub-policies are “Permit” and “NotApplicable” respectively. The condition  $\sigma_p \leq \sigma_n$  means that the firing sequence of  $\sigma_p$  has a shorter length than the sequence  $\sigma_n$ , so the first decision is “Permit”. Note that  $\sigma_p$  and  $\sigma_n$  are firing sequences of the two sub-policies. Hence the final decision is made based not only on the sub-policies decisions but also on their firing sequences.

Corresponding to the Only-one-applicable combiner, the colored Petri net module  $OAC = (\{p_{5e}, p_{5x}\}, T_c, F_c, W_c, M_c, C_c)$  is shown in Figure 6. It contains only two transitions: if there exists only one decision “Permit”  $(1, 0, 0)$  or “Deny”  $(0, 1, 0)$ , or if there are two decisions “NotApplicable”  $(0, 0, 2)$ , the transition  $t_d$  can be fired and the place  $p_{5x}$  gets a token colored in the same way, i.e. accordingly to these decisions. Otherwise transition  $t_n$  is fired and  $p_{5x}$  outputs a decision “NotApplicable”.

By analogy with the XACML combiners, let us call conflict solving combiners the colored Petri nets POC, DOC, FAC, OAC.



**Fig. 5.** First-applicable combiner FAC



**Fig. 6.** Only-one-applicable combiner OAC

**Proposition 5.** *POC, DOC, FAC, OAC are CPNP which are complete, strongly terminating, consistent and confluent.*

*Proof.* For each of POC, DOC, FAC and OAC, there are only two reachable markings, one is the initial marking and another is the exit marking. All the firing sequences contains only one transition in  $T_c$ . It is obvious that they are complete, strongly terminating, consistent and confluent.

In order to formalize the modular composition of the component policies with these conflict solving combiners, we need to introduce firstly more structural combiners for colored Petri nets.

#### 4.2 CPNP-Based Specification of Policy Composition According to PPPA

PPPA is the abbreviation of Property-preserving Petri net process algebra [27,40]. The algebra defines about twenty operators based on PNP and considers the preservation of about twenty system properties. The details of the formal definition of these operators and the proof of the property-preservation results can be found in two PhD theses [27,40].

In order to give some hints about how to apply PPPA to the specification of security policy design, we restrict our attention in this paper to only three logic related composition operators, namely Enable, Choice and Interleave, and to the security-policy related properties, i.e., completeness, termination, consistency and confluence. The application of other operators follows similar ideas.

The Enable composition  $B_1 >> B_2$  models the sequential execution of two processes  $B_1$  and  $B_2$ . That is,  $B_1$  is firstly executed and  $B_2$  is executed after the successful termination of  $B_1$ . However, if  $B_1$  does not exit successfully,  $B_2$  will never be activated.

**Definition 10.** (*Enable*) (Fig. 7) For two processes  $B_i = (P_i, T_i, F_i, W_i, M_{i0}, C_i, P_{ie}, P_{ix})$  ( $i = 1, 2$ ), their composition by Enable, denoted  $B_1 >> B_2$ , is defined as the process  $B = (P, T, F, W, M_0, C, p_e, p_x)$ , where  $P = P_1 \cup P_2$ ,  $p_e = p_{1e}$ ,  $p_x = p_{2x}$  and  $p_{2e}$  is merged with  $p_{1x}$ ;  $T = T_1 \cup T_2$ ;  $F = F_1 \cup F_2$ ;  $W = W_1 \cup W_2$ ;  $M_0 = M_{10} \cup M_{20}$ ;  $C = C_1 \cup C_2$ .

The Enable composition preserves the properties of its components.

**Proposition 6.** Let  $B$  be the policy obtained from two sub-policies  $B_1$  and  $B_2$  by applying the composition operator Enable. Then,

1.  $B$  is complete iff both  $B_1$  and  $B_2$  are complete.
2.  $B$  is strongly terminating iff both  $B_1$  and  $B_2$  are strongly terminating;  $B$  is weakly terminating iff  $B_1$  and  $B_2$  are weakly terminating.
3. If both  $B_1$  and  $B_2$  are consistent, then  $B$  is consistent.
4. If both  $B_1$  and  $B_2$  are confluent, then  $B$  is confluent.

*Proof.* For each firing sequence  $\sigma$  in  $B$ , it is either a firing sequence in  $B_1$  or a union of a sequence  $\sigma_1$  in  $B_1$  and a sequence  $\sigma_2$  in  $B_2$ , where  $M_{1e}[B_1, \sigma_1]M_{1x}$  in  $B_1$ . The remaining part of the proof is trivial.

The Choice composition  $B_1[]B_2$  models the arbitrary selection for execution between two processes  $B_1$  and  $B_2$ .

**Definition 11.** (Choice) (Fig. 7) For two processes  $B_i = (P_i, T_i, F_i, W_i, M_{i0}, C_i, p_{ie}, p_{ix})$  ( $i = 1, 2$ ), their composition by Choice, denoted  $B_1[]B_2$ , is defined as the process  $B = (P, T, F, W, M_0, C, p_e, p_x)$ , where  $P = P_1 \cup P_2$ ,  $p_e$  is the place merging  $p_{1e}$  and  $p_{2e}$ ,  $p_x$  is the place merging  $p_{1x}$  and  $p_{2x}$ ;  $T = T_1 \cup T_2$ ;  $F = F_1 \cup F_2$ ;  $W = W_1 \cup W_2$ ;  $M_0 = M_{10} \cup M_{20}$ ;  $C = C_1 \cup C_2$ .

In general, when  $B_1$  and  $B_2$  specify security policies,  $B_1[]B_2$  is not consistent nor confluent, even when both  $B_1$  and  $B_2$  are.

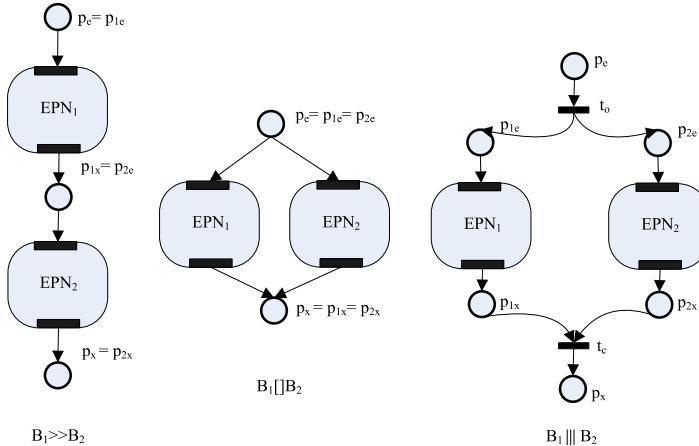
**Proposition 7.** Let the policy  $B$  be obtained from two sub-policies  $B_1$  and  $B_2$  by applying the composition operator Choice. The following results hold:

1.  $B$  is complete iff  $B_1$  and  $B_2$  are complete;
2.  $B$  is strongly (resp., weakly, properly) terminating iff  $B_1$  and  $B_2$  are strongly (resp., weakly, properly) terminating;
3.  $B$  is consistent if both  $B_1$  and  $B_2$  are consistent and both output a token colored in the same way in their exit places.
4.  $B$  is not always confluent even if both  $B_1$  and  $B_2$  are confluent.  $B$  is confluent if both  $B_1$  and  $B_2$  are properly terminating and output a token colored in the same way in their exit places.

*Proof.* After applying the Choice operator, the control flow is within one of the sub-policies, so the first two properties are trivial. For property 3, although both  $B_1$  and  $B_2$  are consistent, they may output different decisions. Hence  $B$  is not always consistent unless both  $B_1$  and  $B_2$  always output a token colored in the same way in their exit place. For proving property 4, let us suppose that  $M_i \in R(B_i, M_{ie})$ ,  $i = 1, 2$ . Then  $M_i \in R(B, M_e)$ . There is no reachable marking  $M \in R(B, M_1) \cap R(B, M_2)$  except  $M = M_x = M_0 + p_x$ . Hence,  $B$  is not confluent unless both  $B_1$  and  $B_2$  terminate properly and output a token colored in the same way in their exit places.

The Interleave composition  $B_1|||B_2$  models the concurrent but independent execution of two processes  $B_1$  and  $B_2$  with a synchronized exit.

**Definition 12.** (Interleave) (Fig. 7) For two processes  $B_i = (P_i, T_i, F_i, W_i, M_{i0}, C_i, P_{ie}, P_{ix})$  ( $i = 1, 2$ ), their composition by Interleave, denoted  $B_1|||B_2$ , is defined as the process



**Fig. 7.** The policy composition operators according to PPPA

$B = (P, T, F, W, M_0, C, p_e, p_x)$ , where  $P = P_1 \cup P_2 \cup \{p_{1e}, p_{2e}\}$ ,  $p_e$  and  $p_x$  are the newly added entry and exit places, respectively;  $T = T_1 \cup T_2 \cup \{t_0, t_c\}$ , where  $t_0$  and  $t_c$  are newly added transitions;  $F = F_1 \cup F_2 \cup \{(p_e, t_0), (t_0, p_{1e}), (t_0, p_{2e})\} \cup \{(p_{1x}, t_c), (p_{2x}, t_c), (t_c, p_x)\}$ ;  $W = W_1 \cup W_2 \cup \{W((p_e, t_0), W(t_0, p_{1e}), W(t_0, p_{2e}))\} \cup \{W(p_{1x}, t_c), W(p_{2x}, t_c), W(t_c, p_x)\}$ ;  $M_0 = M_{10} \cup M_{20}$ ;  $C = C_1 \cup C_2$ .

The Interleave composition preserves the properties of its components.

**Proposition 8.** Let the policy  $B$  be obtained from two sub-policies  $B_1$  and  $B_2$  by applying the composition operator Interleave. The following results hold:

1.  $B$  is complete iff  $B_1$  and  $B_2$  are complete;
2.  $B$  is strongly (resp., properly) terminating iff  $B_1$  and  $B_2$  are strongly (resp., properly) terminating;  $B$  is weakly terminating iff  $B_1$  and  $B_2$  are weakly terminating;
3.  $B$  is consistent iff  $B_1$  and  $B_2$  are consistent;
4.  $B$  is confluent iff  $B_1$  and  $B_2$  are confluent.

*Proof.* 1.  $B$  is complete iff transition  $t_c$  is firable, i.e., both  $B_1$  and  $B_2$  are complete.

2. Since  $B_1$  and  $B_2$  are executed independently, each firing sequence of  $B$  is a union of sequences of  $B_1$  and  $B_2$ . Property 2 follows easily.

3. If both  $B_1$  and  $B_2$  are consistent, for a request, the output of each sub-policy is always the same, then the output of transition  $t_c$  is unique and  $B$  is consistent. On the other hand, if  $B$  is consistent, the token color in  $p_x$  is unique, correspondingly, each input token for  $t_c$  is unique, i.e., the token color in places  $p_{1x}$  and  $p_{2x}$  should be unique, implying that both  $B_1$  and  $B_2$  are consistent.

4. For any two reachable markings  $M_i = P_{im} + Q_{im}$  in  $B$ , where  $P_{im}, Q_{im}$  are markings in  $B_1$  and  $B_2$  respectively and  $i = 1, 2$ , since both  $B_1$  and  $B_2$  are confluent, there exist  $M'_1 \in R(B_1, P_{1m}) \cap R(B_1, P_{2m})$  and  $M'_2 \in R(B_2, Q_{1m}) \cap R(B_2, Q_{2m})$ . Then  $M = M'_1 + M'_2 \in R(B, M_1) \cap R(B, M_2)$  and  $B$  is confluent.

In the definition of composition with the Interleave operator, the added transition  $t_c$  is actually the key to specify the synchronization between the outputs of the components. In the context of security policies that may return conflicting or inconsistent decisions, the transition  $t_c$  can be refined with an XACML combiner  $COM$  belonging to the set  $\{POC, DOC, FAC, OAC\}$ . This is denoted as  $B_1 \parallel B_2[t_c \leftarrow COM]$ . It is easy to deduce from Propositions 5 and 8 that  $B_1 \parallel B_2[t_c \leftarrow COM]$  preserves the properties of termination, consistency and confluence of its components  $B_1$  and  $B_2$ .

Based on the presented composition operators, a large security policy system can be specified step by step by composing its different modules. The previous propositions help verifying properties of a large system composed with these operators.

### 4.3 An Illustrative Example

Let us now illustrate on an example how to apply the previously introduced operators for composing policy modules.

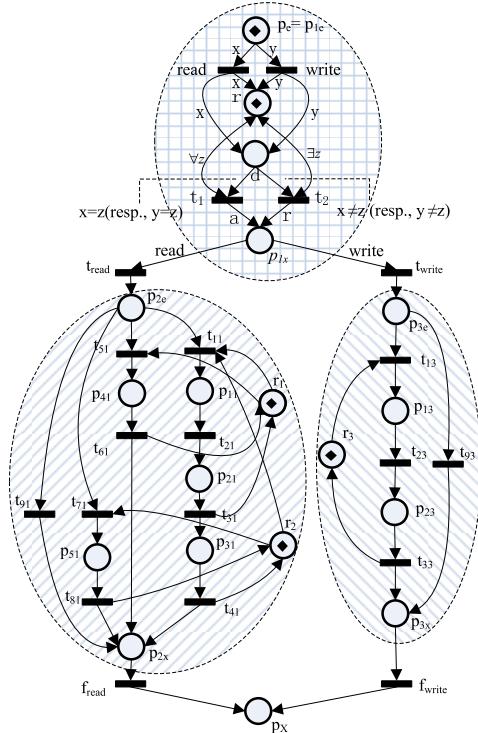
Let us consider a situation where a user requests access to documents belonging to different competitive companies. Such access is granted or denied on the basis on a Chinese Wall Policy [53]; if the user has a reading access to a document, he may print it and/or copy it. If he has a writing access, he may modify it for instance by signing the document.

For the sake of completeness, let us briefly introduce the Chinese Wall Policy CWP. Its aim is to prevent conflict of interest between clients. The objects of the database contain the information related to companies; a company dataset (CD) contains objects related to a single company; a conflict of interest (COI) class contains the datasets of companies in competition. The read and write policies in CWP are as follows. *Read policy*: a subject  $s$  is permitted to read an object  $o$  provided that, either  $s$  reads the objects all in the same CD, or reads the objects in different COIs. In the same COI, the subject cannot read objects in different CDs. *Write policy*: a subject  $s$  is permitted to write an object  $o$  only when  $s$  can read  $o$  and other objects accessible by  $s$  are in the same CD as  $o$ .

The information flow is as follows: a user requests “reading” or “writing” a document according to the Chinese Wall Policy  $B_1$ ; in the first case, once the access decision for reading is obtained, the user can continue processing the document by applying the “printer accessing policy”  $B_2$ ; in the second case, once the access decision for writing is obtained, the user can process the document by applying the “writing access policy”  $B_3$ . For simplicity,  $B_2$  and  $B_3$  just state that whenever the resources (printer, copying machine or document) are available, the corresponding access request is permitted.

The detailed CPNP specification of the global policy is shown in Fig. 8. Let us explain it in more details.

According to the Chinese Wall Policy, each request, either “reading” (specified with color  $x$ ) or “writing” (specified with color  $y$ ), is recorded in order to keep trace of the previous accesses (these records are specified with place  $r$ ). Based on the policy, for a “reading” request, if all the data recorded in place  $r$  belong to the same color, i.e.,  $\forall z, x = z$ , transition  $t_1$  is firable and the output is “(read, accept)”; otherwise if  $\exists z, x \neq z$ , transition  $t_2$  is firable and the output is “(read, reject)”. As for a “writing” request, if  $\exists z \neq y$ , transition  $t_2$  is firable and the output is “(write, reject)”; otherwise,  $\forall z, y = z$ ,



**Fig. 8.** CPNP specification of a component-based policy

transition  $t_1$  is firable and the output is “(write, accept)”. These output decisions are reading or writing requests for respectively  $B_2$  and  $B_3$ .

In case of a reading request, transition  $t_{read}$  is firable and, provided the decision of  $B_1$  was “(read, accept)”, the user can request copying the document (firing transition  $t_{71}$ ), or printing the document (firing transition  $t_{51}$ ), or printing and copying (firing transition  $t_{11}$ ). Once there exists an available copying machine (i.e. when  $r_2$  is marked), the user can copy. After copying, transition  $t_{81}$  is fired and the copying machine is released; once there exists an available printer ((i.e. when  $r_1$  is marked), the user can print. After printing, transition  $t_{61}$  is fired and the printer is released; once there exist available printers and copying machines, the user can print and copy; then transitions  $t_{31}$  and  $t_{41}$  are fired and both resources are released. If the decision of  $B_1$  was “(read, reject)”, transition  $t_{91}$  is fired. The final transition  $f_{read}$  just outputs “OK”, meaning that the request has been handled.

In case of a writing request, transition  $t_{write}$  is firable and, provided the decision of  $B_1$  was “(write, accept)”, the user can write the document. When the document is available (i.e. when place  $r_3$  is marked),  $t_{13}$  is firable and the user can process writing the document ( $t_{23}$  corresponds to the operation of writing); after writing, the document is released and updated ( $t_{33}$  is fired). If the decision of  $B_1$  was “(write, reject)”, transition  $t_{93}$  is fired. The final transition  $f_{write}$  just outputs “OK”, meaning that the request has been handled.

The composed policy (Fig. 8) consists of three sub-policies specified with  $B_1$ ,  $B_2$  and  $B_3$  respectively, which are combined by applying Enable and Choice operators based on the information flow. It is easy to verify that all three sub-policies are correct, i.e., complete, terminating, consistent and confluent. Since  $B_2$  and  $B_3$  always output the same decision “OK”, based on Propositions 6 and 7, the composed policy model shown in Fig. 8 is correct too.

## 5 Conclusion

In this paper, we mainly consider security policies that give access permissions. Indeed the newly defined colored Petri net process fits very well to policies that receive access requests and deliver access decisions. However this can be easily extended to more general security policies where the entry and exit places correspond to input and output interfaces with the environment.

Our work here is rather focussed on modular specification of security policies. Policy composition operators are specified and property-preserving results are stated for verification. This technology is suitable for general security policy design, especially for large and complex security systems. In a real-life software system, the system requirements may be changeable and the security policy is complex. Hence, more policy composition operators should be available for the specification. For instance, in [28], this modular specification and verification technology is applied for the Chinese wall policy design. This example needs to introduce another operator, called Disable composition, that can connect a place in one of the component to a transition in the other. Not surprisingly the properties of confluence and consistency are more difficult to preserve in this case.

In [28], we also explored policy compositions such that the integrated policy is capable of handling resources sharing, simultaneously executing operations and embedding sub-policies into super-policies in multiple heterogeneous systems. Again it is interesting to state under which conditions the global policy preserves the fundamental policy properties, i.e., completeness, termination, consistency and confluence, and satisfy policy autonomy and security principles that are required for secure interoperation.

In future work, we expect to synthesize from the study of different approaches to the composition of security policies, a set of useful composition operators that preserve policy properties and provide the basis of a domain specific language.

Another outcome of this paper could be a better understanding of respective advantages of Petri nets and rewriting systems formalisms. There are already close connections between these two approaches which have been settled. Rewriting logic has been proposed as a unifying framework for a wide range of Petri nets models in [48], where different kinds of Petri nets are translated in rewriting logic. Briefly speaking, markings are expressed as data types with token constructors and transitions are rewrite rules. The set of reachable markings corresponds to the rewriting derivations. As previously mentioned, the property of confluence defined for Petri nets has been related to confluence in rewriting systems in [37]. For computational complexity, both approaches face the state exploding problem if reachability analysis is applied. However, Petri nets and rewrite systems differ in their ability to model distinct aspects of security policies.

Petri nets fit for those policies specified by their activities and the transitions involve resources and time, although formalization of different resources is somehow involved. Control flow is made explicit through tokens. Rewrite systems formalization is backed to rewriting logic and calculus, that provide easy way to reason about consistency, termination and well-definedness. Control is expressible through strategies in a declarative and explicit way. This paper is an attempt to deeper study whether compositionality and modularity, well-expressed and explored in the rewriting approach, are also relevant for Petri net specifications of security policies. Conversely, we think that constructs borrowed from Petri net theory and studied in this paper, can actually be expressed in the strategic rewriting approach. However further work is needed to formally validate this intuition.

## References

1. Astesiano, E., Bidoit, M., Kirchner, H., Krieg-Brückner, B., Mosses, P.D., Sannella, D., Tarlecki, A.: CASL: the common algebraic specification language. *Theor. Comput. Sci.* 286(2), 153–196 (2002)
2. Barker, S., Fernández, M.: Term rewriting for access control. In: DBSec, pp. 179–193 (2006)
3. Bauer, L., Ligatti, J., Walker, D.: Composing security policies with polymer. In: PLDI, pp. 305–314 (2005)
4. Bell, D., LaPadula, L.: Secure computer systems: A mathematical model. *Journal of Computer Security* ii. 4(2/3), 229–263 (1996)
5. Bertino, E., Bettini, C., Ferrari, E., Samarati, P.: An access control model supporting periodicity constraints and temporal reasoning. *ACM Trans. Database Syst.* 23(3), 231–285 (1998)
6. Bertolissi, C., Fernández, M.: An algebraic-functional framework for distributed access control. In: International Conference on Risks and Security of Internet and Systems (CRISIS 2008), Tozeur, Tunisia. Proceedings IEEE Xplorer to appear (2008)
7. Bertolissi, C., Fernández, M.: A rewriting framework for the composition of access control policies. In: Proceedings of PPDP 2008, Valencia. ACM Press, New York (2008)
8. Biba, K.: Integrity considerations for secure computer systems. Technical Report TR-3153, Mitre, Bedford, MA (1975)
9. Bonatti, P., De Capitani di Vimercati, S., Samarati, P.: An algebra for composing access control policies. *ACM Trans. Inf. Syst. Secur.* 5(1), 1–35 (2002)
10. Bonatti, P., Olmedilla, D.: Driving and monitoring provisional trust negotiation with metapolices. In: Proceedings IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY). IEEE Society, Los Alamitos (2005)
11. Bonatti, P., Samarati, P.: A uniform framework for regulating service access and information release on the web. *Journal of Computer Security* 10(3), 241–272 (2002)
12. Brewer, D.F.C., Nash, M.J.: The chinese wall security policy. In: Proc. IEEE Symposium on Security and Privacy, pp. 206–214 (1989)
13. Bruns, G., Dantas, D., Huth, M.: A simple and expressive semantic framework for policy composition in access control. In: FMSE 2007: Proceedings of the 2007 ACM workshop on Formal methods in security engineering, pp. 12–21. ACM Press, New York (2007)
14. Bruns, G., Huth, M.: Access-control policies via belnap logic: Effective and efficient composition and analysis. In: 21st IEEE Computer Security Foundations Symposium (CSF), pp. 163–176. IEEE Computer Society Press, Los Alamitos (2008)

15. Campbell, R., Al-Muhtadi, J., Naldurg, P., Sampemane, G., Mickunas, M.D.: Towards security and privacy for pervasive computing. In: Okada, M., Pierce, B.C., Scedrov, A., Tokuda, H., Yonezawa, A. (eds.) ISSS 2002. LNCS, vol. 2609, pp. 1–15. Springer, Heidelberg (2003)
16. Cholvy, L., Cuppens, F.: Analyzing consistency of security policies. In: IEEE Symposium on Security and Privacy, pp. 103–112 (1997)
17. Choppy, C., Petrucci, L.: Towards a methodology for modeling with Petri nets. In: Proc. Workshop on Practical Use of Coloured Petri Nets (CPN 2004), pp. 39–56 (2004)
18. Cirstea, H., Moreau, P., Santana de Oliveira, A.: Rewrite based specification of access control policies. In: Dougherty, D., Escobar, S. (eds.) Security and Rewriting Techniques, 3rd International Workshop SecRet 2008. Electronic Notes in Theoretical Computer Science. Elsevier, Amsterdam (2008)
19. Cuppens, F., Cuppens-Boulahia, N., Sans, T.: A security model with non-atomic actions and deadlines. In: CSFW, pp. 186–196. IEEE Society, Los Alamitos (2005)
20. de Oliveira, A.S., Wang, E.K., Kirchner, C., Kirchner, H.: Weaving rewrite-based access control policies. In: FMSE 2007: Proceedings of the 2007 ACM workshop on Formal methods in security engineering. ACM, New York (2007)
21. Deng, Y., Wang, J.C., Tsai, J., Beznosov, K.: An approach for modeling and analysis of security system architectures. *IEEE Transactions on Knowledge and Data Engineering* 15(5), 1099–1119 (2003)
22. Dougherty, D.J., Fisler, K., Krishnamurthi, S.: Specifying and reasoning about dynamic access-control policies. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 632–646. Springer, Heidelberg (2006)
23. Dougherty, D.J., Kirchner, C., Kirchner, H., Santana de Oliveira, A.: Modular access control via strategic rewriting. In: Biskup, J., López, J. (eds.) ESORICS 2007. LNCS, vol. 4734, pp. 578–593. Springer, Heidelberg (2007)
24. Habib, L., Jaume, M., Morisset, C.: A formal comparison of the bell & lapadula and rbac models. In: Fourth International Symposium on Information Assurance and Security (IAS 2008), pp. 3–8. IEEE Computer Society Press, Los Alamitos (2008)
25. Halpern, J.Y., Weissman, V.: Using first-order logic to reason about policies. In: CSFW, pp. 187–201 (2003)
26. Harrison, M.A., Ruzzo, W.L., Ullman, J.D.: Protection in operating systems. *Commun. ACM* 19(8), 461–471 (1976)
27. Huang, H.J.: Enhancing the Property-Preserving Petri Net Process Algebra for Component-based System Design (with Application to Designing Multi-agent Systems and Manufacturing Systems). PhD thesis, Department of Computer Science, City University of Hong Kong (2004)
28. Huang, H.J., Kirchner, H.: Modular security policy design based on extended Petri nets. Technical Report inria-00396924, INRIA (2009), <http://hal.inria.fr/inria-00396924/fr/>
29. Jajodia, S., Samarati, P., Sapino, M.L., Subrahmanian, V.S.: Flexible support for multiple access control policies. *ACM Trans. Database Syst.* 26(2), 214–260 (2001)
30. Jarvis, B., Jain, L.: Trust in LORA: Towards a formal definition of trust in BDI agents. In: Gabrys, B., Howlett, R.J., Jain, L.C. (eds.) KES 2006. LNCS (LNAI), vol. 4252, pp. 458–463. Springer, Heidelberg (2006)
31. Jaume, M., Morisset, C.: Towards a formal specification of access control. In: Degano, P., Kusters, R., Vigano, L., Zdancewic, S. (eds.) Proceedings of the Joint Workshop on Foundations of Computer Security and Automated Reasoning for Security Protocol Analysis FCS-ARSPA 2006, pp. 213–232 (2006)

32. Jensen, K.: Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use, vol. 1. Springer, Berlin (1997)
33. Juszczyszyn, K.: Verifying enterprise's mandatory access control policies with coloured Petri nets. In: Proceedings of the 12th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (2003)
34. Kalam, A., Baida, R., Balbiani, P., Benferhat, S., Cuppens, F., Deswart, Y., Miege, A., Saurel, C., Trouessin, G.: Organization based access control. In: Proceedings IEEE 4th International Workshop on Policies for Distributed Systems and Networks (POLICY), pp. 120–131. IEEE Society, Los Alamitos (2003)
35. Kirchner, C., Kirchner, H., Santana de Oliveira, A.: Analysis of rewrite-based access control policies. In: Dougherty, D., Escobar, S. (eds.) Security and Rewriting Techniques, 3rd International Workshop Secret 2008, Electronic Notes in Theoretical Computer Science. Elsevier, Amsterdam (2008)
36. Knorr, K.: Dynamic access control through Petri net workflows. In: Proceedings of 16th Annual Conference on Computer Security Applications, pp. 159–167 (2000)
37. Leahu, I., Tiplea, F.: The confluence property for Petri nets and its applications. In: Proceedings of the 8th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (2006)
38. Lee, A.J., Boyer, J.P., Olson, L., Gunter, C.A.: Defeasible security policy composition for web services. In: Winslett, M., Gordon, A.D., Sands, D. (eds.) FMSE, pp. 45–54. ACM Press, New York (2006)
39. Li, N., Mitchell, J.C.: Datalog with constraints: A foundation for trust management languages. In: PADL, pp. 58–73 (2003)
40. Mak, W.: Verifying Property Preservation for Component-based Software Systems (A Petri-net Based Methodology). PhD thesis, Department of Computer Science, City University of Hong Kong (2001)
41. Mortensen, K.: Automatic code generation method based on coloured Petri net models applied on an access control system. In: Nielsen, M., Simpson, D. (eds.) ICATPN 2000. LNCS, vol. 1825, pp. 367–386. Springer, Heidelberg (2000)
42. Moses, T.: Extensible access control markup language (XACML) version 2.0. Technical report, OASIS (February 2005)
43. Mosses, P.D.: Component-based description of programming languages. In: Visions of Computer Science, Electronic Proceedings, pp. 275–286. BCS (2008)
44. Murata, T.: Petri nets: Properties, analysis, and applications. Proceedings of IEEE 77(4), 541–580 (1985)
45. Santana de Oliveira, A.: Réécriture et modularité pour les politiques de sécurité. PhD thesis, UHP Nancy 1 (2008)
46. Shafiq, B., Masood, A., Joshi, J., Ghafoor, A.: A role-based access control policy verification framework for real-time systems. In: Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (2005)
47. Shandu, R., Coyne, E., Feinstein, H., Youman, C.: Role-based access control models. IEEE Computer 29(2), 38–47 (1996)
48. Stehr, M.-O., Meseguer, J., Olveczky, P.C.: Rewriting logic as a unifying framework for Petri nets. In: Ehrig, H., Juhás, G., Padberg, J., Rozenberg, G. (eds.) APN 2001. LNCS, vol. 2128, pp. 250–303. Springer, Heidelberg (2001)
49. Tiplea, F., Jucan, T., Masalagiu, C.: Term rewriting systems and Petri nets. Analele Stiintifice ale Universitatii Al. I. Cuza 34(4), 305–317 (1988)

50. Tschantz, M.C., Krishnamurthi, S.: Towards reasonability properties for access-control policy languages. In: Ferraiolo, D.F., Ray, I. (eds.) SACMAT, pp. 160–169. ACM Press, New York (2006)
51. Verma, R., Rusinowitch, M., Lugiez, D.: Algorithms and reductions for rewriting problems. *Fundamental Informatics* 46(3), 257–276 (2001)
52. Wijesekera, D., Jajodia, S.: A propositional policy algebra for access control. *ACM Trans. Inf. Syst. Secur.* 6(2), 286–325 (2003)
53. Zhang, Z., Hong, F., Liao, J.: Modeling chinese wall policy using colored Petri nets. In: Proceedings of the 6th IEEE International Conference on Computer and Information Technology (2006)
54. Zhang, Z., Hong, F., Xiao, H.: Verification of strict integrity policy via Petri nets. In: Proceedings of the International Conference on Systems and Networks Communication (2006)

# Order-Sorted Parameterization and Induction

José Meseguer

University of Illinois at Urbana-Champaign

**Abstract.** Parameterization is one of the most powerful features to make specifications and declarative programs modular and reusable, and our best hope for scaling up formal verification efforts. This paper studies order-sorted parameterization at three different levels: (i) its mathematical semantics; (ii) its operational semantics by term rewriting; and (iii) the inductive reasoning principles that can soundly be used to prove properties about such specifications. It shows that achieving the desired properties at each of these three levels is a considerably subtler matter than for many-sorted specifications, but that such properties can be attained under reasonable conditions.

## 1 Introduction

Peter Mosses and I met for the first time in the Summer of 1976. It was the beginning of a life-long friendship. I was in the UK presenting a paper at a category theory meeting in Sussex and, before visiting Joseph Goguen and Rod Burstall in Edinburgh, I spent some days in Oxford visiting Peter to try to become better acquainted with denotational semantics. Peter was very hospitable and found a place for me to work next to his in the quaint Victorian house that was then the home of Oxford’s Programming Research Group in Bainbury Road.

Peter gave me the best reference then available: a mimeographed version of Strachey and Milne’s book [54], which was about to be published. I struggled with the semantic definitions in the book, which became harder and harder to understand as more features were added to the language. In hindsight, the problem was not only my ignorance, but also the utter lack of *modularity* in those definitions, which were reformulated over and over again with what I like to call new “monkeys” (stores, environments, continuations, input-output, and so on) in a veritable *tour de force*.

I see the quest for solving problems of modularity as one of the main threads, perhaps the *leitmotiv*, of Peter’s many seminal contributions. In different ways, besides working of course on other topics, Peter has been doggedly attacking this problem over the years and giving us newer and more comprehensive solutions. I should mention in this regard his work on Abstract Semantic Algebras [41], Action Semantics [40], Unified Algebras [44,42], and his more recent and beautiful work on Modular Operational Semantics [47], all of which provide solutions to the modularity problem for semantic definitions of programming languages. Furthermore, Peter has also made key contributions to the modularity of formal specification languages, spear-heading the entire CASL language effort [46,3].

For honoring Peter in this Festschrift I have chosen a topic that is close to Peter's own research in several ways, namely, parameterized order-sorted specifications. The whole point of such specifications is of course the great increase in modularity and reusability that they provide. Indeed, Peter has worked not only in languages like CASL that support a variant of parameterized order-sorted specifications and on many topics in algebraic specification: he has made seminal contributions to variants of order-sortedness such as his unified algebras [43,42,44], and his survey on the use of sorts in algebraic specifications [45].

At the technical level, my main interests in this topic are both conceptual and pragmatic. Conceptually, I view this work as completing, at least in part, the new semantics of order-sorted algebra that I proposed in [38] because of what I felt were limitations in both the approach that Goguen and I proposed in [25], and the approach by Poigné [50] (see Section 2). But my interests are not just about the *mathematical* semantics of parameterized order-sorted specifications, but also about their *operational* semantics by term rewriting. From this point of view, a parameterized order-sorted specification in a language like OBJ [26], CafeOBJ [18], Maude [11], or CASL [46], is a *program* that we want to execute and to reuse in many different contexts. Therefore, the correctness of its executability is just as important as that of its mathematical semantics. This means that the *stability under pushouts* of operational properties such as confluence, termination, and sort-decreasingness should be studied in as much depth as, say, the stability under pushouts of persistence. Yet another important concern is *inductive reasoning*. The point is that the great modularity and reusability features of parameterized specifications should also accrue to their inductive theorems, a key feature if we are going to have any hope of scaling up formal verification efforts to large systems. Furthermore, I am particularly interested in these problems in connection with the Maude specification language [11], both in terms of the language itself, which supports parameterized specifications in order-sorted equational logic, membership equational logic, and rewriting logic, and in terms of its formal reasoning methods and tools.

Perhaps the main thing to say about the sections to come is that they show that, unlike the case of many-sorted equational specifications for which parameterization issues are very well understood and relatively simple, order-sorted parameterization issues are considerably subtler. In particular, the matter of *stability under pushouts* is nontrivial and may fail to hold for different properties. In some sense I show that the mathematical and operational semantics aspects help each other to ensure that desired properties of parameterized specifications are indeed stable. But even when an order-sorted parameterized specification is stable under pushouts in all the ways we want, the matter of its *inductive theorems* remains more subtle than for many-sorted specifications, since some of the inductive theorems that hold for a parameterized specification may fail to hold for pushout instances of it. Nevertheless, it is indeed possible to find reasonable conditions so that parameterized order-sorted specifications behave as we would like them to, both mathematically and operationally, and so that we can reason in a sound way about their inductive properties.

The paper is organized as follows. I summarize the main concepts of order-sorted algebra needed in the rest of the paper in Section 2, discuss pushouts of signatures and failures with amalgamation in Section 3, and give examples showing that both persistence and convergence of rewrite rules can fail to be stable under pushouts in Section 4. This chain of negative results is followed by a chain of positive ones. In Section 5 I give conditions under which convergence of rewrite rules is stable under pushouts. And in Sections 6 and 7 I connect convergence issues with persistence to arrive at conditions under which parameterized free constructor specifications, and parameterized specifications that are sufficiently complete with respect to a free constructor subspecification, are persistent. Inductive properties of order-sorted parameterized specifications are then studied in detail in Sections 8 and 9. Related work and conclusions are discussed in Section 10.

## 2 Order-Sorted Algebra Revisited

In [38] I studied the main model-theoretic properties of membership equational logic ( $MEqtl$ ), including the question of how exactly it could be viewed as a conservative generalization of order-sorted equational logic. This led me to a critical reconsideration of the foundations of order-sorted algebra (OSA). The point is that several non-equivalent axiomatizations of OSA had been proposed, including the ones by Goguen and Meseguer [25] ( $OSA^{GM}$ ) and by Poigné [50] ( $OSA^P$ ), each with its own advantages and disadvantages. I found both  $OSA^{GM}$  and  $OSA^P$  unsatisfactory in some respects and proposed a new version  $OSA^R$  that combined their respective advantages but avoided their drawbacks.

I summarized in [38] the advantages of  $OSA^R$  in tabular form as follows.

	$OSA^{GM}$	$OSA^P$	$OSA^R$
Contains MSA as special case	Yes	No	Yes
Term algebras initial with minimal requirements	No	Yes	Yes
Pushouts of theories exist in general	No	Yes	Yes
Ad-hoc overloading possible	Yes	No	Yes
Terms of different sorts allowed in equations	Yes	No	Yes
Natural conservative extension map to $MEqtl$	No	Yes	Yes

In the rest of this section I recall the basic definitions and some of the key results about  $OSA^R$  that are needed in the rest of the paper. Some details that can be found in [38] are omitted, but basic notions about order-sorted term rewriting and unification not contained in [38] are included.

**Definition 1.** An order-sorted signature is a triple  $\Sigma = (S, \leq, \Sigma)$  with  $(S, \leq)$  a poset and  $(S, \Sigma)$  a many-sorted signature.

We denote by  $\hat{S} = S/\equiv_{\leq}$  the set of connected components of  $(S, \leq)$ , which is the quotient of  $S$  under the equivalence relation  $\equiv_{\leq}$  generated by  $\leq$ . The equivalence  $\equiv_{\leq}$  can be extended to sequences in the usual way.

An order-sorted signature is called sensible if for any two operators  $f : w \rightarrow s, f : w' \rightarrow s'$ , with  $w$  and  $w'$  of same length, we have  $w \equiv_{\leq} w' \Rightarrow s \equiv_{\leq} s'$ .  $\square$

The notion of a sensible signature is a minimal syntactic requirement to avoid excessive ambiguity. It is a much weaker requirement than preregularity [25]. A signature may be sensible and still allow so-called ad-hoc overloading of operators, where the same function symbol is used with typings that are unrelated in the subsort ordering. Indeed, we can have  $f : w \rightarrow s, f : w' \rightarrow s'$ , with  $w$  and  $w'$  of same length but with  $w \not\equiv_{\leq} w'$ .

**Notation.** For connected components  $[s_1], \dots, [s_n], [s] \in \hat{S}$

$$f_{[s]}^{[s_1]\dots[s_n]} = \{f : s'_1 \dots s'_n \rightarrow s' \mid s'_i \in [s_i] \quad 1 \leq i \leq n, s' \in [s]\}$$

denotes the family of “subsort polymorphic” operators with name  $f$  for those components.

**Definition 2.** Signature morphisms  $H : (S, \leq, \Sigma) \rightarrow (S', \leq', \Sigma')$  are many-sorted signature morphisms  $H : (S, \Sigma) \rightarrow (S', \Sigma')$  with  $H : (S, \leq) \rightarrow (S', \leq')$  monotonic and such that they preserve subsort overloading, that is,  $H$  must map the operators in each subsort polymorphic family  $f_{[s]}^{[s_1]\dots[s_n]}$  to a subset of the operators in the subsort polymorphic family  $H(f)_{[H(s)]}^{[H(s_1)]\dots[H(s_n)]}$ .  $\square$

**Theorem 1.** [38] The category **OSSign** of order-sorted signatures and signature morphisms is cocomplete.

**Definition 3.** For  $\Sigma = (S, \leq, \Sigma)$  an order-sorted signature, an order-sorted  $\Sigma$ -algebra  $A$  is a many-sorted  $(S, \Sigma)$ -algebra  $A$  such that

- whenever  $s \leq s'$ , then we have  $A_s \subseteq A_{s'}$ , and
- whenever  $f : w \rightarrow s, f : w' \rightarrow s'$  in  $f_{[s]}^{[s_1]\dots[s_n]}$  (with  $w = s_1 \dots s_n$ ), and  $\bar{a} \in A^w \cap A^{w'}$ , then we have  $A_{f:w \rightarrow s}(\bar{a}) = A_{f:w' \rightarrow s'}(\bar{a})$ .

An order-sorted  $\Sigma$ -homomorphism  $h : A \rightarrow B$  is a many-sorted  $(S, \Sigma)$ -homomorphism such that whenever  $[s] = [s']$  and  $a \in A_s \cap A_{s'}$ , then we have  $h_s(a) = h_{s'}(a)$ . We call  $h$  injective, resp. surjective, resp. bijective, iff for each  $s \in S$   $h_s$  is injective, resp. surjective, resp. bijective. We call  $h$  an isomorphism if there is another order-sorted  $\Sigma$ -homomorphism  $g : B \rightarrow A$  such that for each  $s \in S$ ,  $h_s \circ g_s = 1_{A_s}$ , and  $g_s \circ h_s = 1_{B_s}$ . This defines a category **OSAlg** $_{\Sigma}$ .  $\square$

Every  $\Sigma$ -isomorphism is clearly bijective. However, unless we assume that each connected component has a top sort, it is easy to exhibit signatures where some bijective homomorphisms fail to be isomorphisms.

Notice that, given an order-sorted  $\Sigma$ -algebra  $A$ , if we denote by  $A_{[s]} = \bigcup_{s' \in [s]} A_{s'}$  for each  $[s] \in \hat{S}$ , then each  $f_{[s]}^{[s_1]\dots[s_n]}$  determines a single partial function

$$A_{f_{[s]}^{[s_1]\dots[s_n]}} : A_{[s_1]} \times \dots \times A_{[s_n]} \rightharpoonup A_{[s]} .$$

Similarly, an order-sorted  $\Sigma$ -homomorphism  $h : A \rightarrow B$  determines an  $\hat{S}$ -family of functions

$$\{h_{[s]} : A_{[s]} \rightarrow B_{[s]}\}_{[s] \in \hat{S}} .$$

**Theorem 2.** [38] *The category  $\mathbf{OSAlg}_\Sigma$  has an initial algebra. Furthermore, if  $\Sigma$  is sensible, then the term algebra  $T_\Sigma$  with*

- if  $a : \lambda \rightarrow s$  then  $a \in T_{\Sigma,s}$ ,
- if  $t \in T_{\Sigma,s}$  and  $s \leq s'$  then  $t \in T_{\Sigma,s'}$ ,
- if  $f : s_1 \dots s_n \rightarrow s$  and  $t_i \in T_{\Sigma,s_i} \ 1 \leq i \leq n$ , then  $f(t_1, \dots, t_n) \in T_{\Sigma,s}$ ,

is initial.

$T_\Sigma(X)$  is the algebra of terms with variables  $X = \{X_s\}_{s \in S}$ , where  $s \neq s' \Rightarrow X_s \cap X_{s'} = \emptyset$ . A  $\Sigma$ -equation  $\forall X. t = t'$  must have  $t \in T_\Sigma(X)_s, t' \in T_\Sigma(X)_{s'}$ , with  $[s] = [s']$ . An *order-sorted theory* is a pair  $T = (\Sigma, E)$ , with  $\Sigma$  an order-sorted signature, and  $E$  a set of  $\Sigma$ -equations, which in general may be conditional. Satisfaction of a  $\Sigma$ -equation  $\forall X. t = t'$  by an order-sorted  $\Sigma$ -algebra  $A$ , written  $A \models \forall X. t = t'$ , is defined, and extended to conditional equations, in the usual way. The category  $\mathbf{OSAlg}_T$  is then the full subcategory of  $\mathbf{OSAlg}_\Sigma$  determined by all those algebras that satisfy the equations in  $T$ .

Given an order-sorted theory  $T = (\Sigma, E)$ , the paper [38] gives a sound and complete inference system to derive all the equational theorems of the theory  $T$  under the above  $\text{OSA}^R$  semantics for order-sorted algebras and homomorphisms.

A signature morphism  $H : (S, \leq, \Sigma) \rightarrow (S', \leq', \Sigma')$  induces a forgetful functor

$$U_H : \mathbf{OSAlg}_{\Sigma'} \rightarrow \mathbf{OSAlg}_\Sigma ,$$

with  $U_H(A')_s = A'_{H(s)}$  and  $U_H(A')_f = A'_{H(f)}$  for each algebra  $A' \in \mathbf{OSAlg}_{\Sigma'}$ .

**Definition 4.** Given two order-sorted theories  $T = (\Sigma, E)$  and  $T' = (\Sigma', E')$ , a theory morphism  $H : T \rightarrow T'$  is a signature morphism  $H : \Sigma \rightarrow \Sigma'$  such that for each  $\Sigma$ -equation  $\forall x_1 : s_1, \dots, x_n : s_n. t = t'$  in  $E$  we have  $E' \models \forall x_1 : H(s_1), \dots, x_n : H(s_n). H(t) = H(t')$ , where  $H(t)$  is the obvious translation of terms induced by  $H$ , that is,  $H(x : s) = x : H(s)$ , and  $H(f(t_1, \dots, t_n)) = H(f)(H(t_1), \dots, H(t_n))$ . For any theory morphism  $H : T \rightarrow T'$  the forgetful functor  $U_H$  restricts to a forgetful functor

$$U_H : \mathbf{OSAlg}_{T'} \rightarrow \mathbf{OSAlg}_T .$$

□

The institution [22]  $\text{OSEql}^R$ , whose models are order-sorted algebras (in the  $\text{OSA}^R$  sense), and whose sentences are conditional universally quantified equations, is *liberal*, that is, for any theory morphism  $H : (S, \leq, \Sigma, \Gamma) \rightarrow (S, \leq', \Sigma', \Gamma')$  the forgetful functor  $U_H : \mathbf{OSAlg}_{T'} \rightarrow \mathbf{OSAlg}_T$  has a left adjoint

$$F_H : \mathbf{OSAlg}_T \rightarrow \mathbf{OSAlg}_{T'}$$

In particular, this ensures the existence of *initial and free algebras* in the category  $\mathbf{OSAlg}_T$  for any order-sorted equational theory  $T$ . I give a more detailed

description of the  $F_H$  construction for the case when  $H$  is a theory inclusion in Section 6.

Given a set  $E$  of (implicitly universally quantified) order-sorted equations such that for each  $t = t'$  in  $E$  we have  $\text{vars}(t') \subseteq \text{vars}(t)$ , we can use them as *rewrite rules* from left to right to obtain a more efficient form of equational deduction. That is, given a term  $u \in T_\Sigma(X)$ , the free order-sorted  $\Sigma$ -algebra on variables  $X$ , we define the *one-step* rewrite relation  $u \rightarrow_E v$  iff there is a position  $p$  in  $u$  and a sort-preserving substitution  $\sigma : \text{vars}(t) \rightarrow T_\Sigma(X)$  such that: (i)  $u_p = \sigma(t)$ , (ii)  $v = u[\sigma(t')]_p$ , and (iii)  $v \in T_\Sigma(X)$ , where, as usual, a position  $p$  in  $u$  is specified as a sequence of natural numbers which determines a path from the root when  $u$  is viewed as a tree,  $u_p$  denotes the subterm at that position, and  $u[w]_p$  denotes the term obtained by replacing  $u_p$  by  $w$  in  $u$  at position  $p$ .

Condition (iii), namely, the well-formedness of  $v$  as an order-sorted term is not automatic: it is easy to give  $\Sigma$ -equations  $t = t'$  such that  $v = u[\sigma(t')]_p$  is not well-formed. However, condition (iii) is guaranteed if the equations  $E$  are *sort-decreasing*, meaning that if  $t = t'$  is in  $E$ , then for each sort-preserving substitution  $\sigma$  and sort  $s$  such that  $\sigma(t)$  has sort  $s$ , then  $\sigma(t')$  also has sort  $s$ . Sort-decreasingness is easy to check syntactically by considering renamings of variables that lower the sorts of the variables in  $\text{vars}(t)$ .

As usual, we denote by  $\rightarrow_E^*$  the reflexive-transitive closure of the one-step rewrite relation  $\rightarrow_E$ . We say that the equations  $E$  are *terminating* iff  $\rightarrow_E$  is a well-founded relation, so that there cannot be any infinite sequences of one-step rewrite steps. We say that the equations  $E$  are *confluent* iff whenever we have  $t \rightarrow_E^* u$  and  $t \rightarrow_E^* v$  we can always find a well-formed term  $w$  such that  $u \rightarrow_E^* w$  and  $v \rightarrow_E^* w$ . A useful property about confluent and sort-decreasing equations for equational order-sorted deduction is their completeness in the following sense:

**Theorem 3.** [33] *If the equations  $E$  are confluent and sort-decreasing as rewrite rules, then for any two terms  $u, v \in T_\Sigma(X)$ , we have  $E \models \forall(\text{vars}(u) \cup \text{vars}(v)). u = v$  iff there exists a term  $w$  such that  $u \rightarrow_E^* w$  and  $v \rightarrow_E^* w$ .*

In particular, if the equations  $E$  are *convergent*, that is: (i) confluent, (ii) sort-decreasing, and (iii) terminating, then equality modulo  $E$  becomes *decidable* by rewriting, since we can rewrite  $u$  and  $v$  “to the bitter end” and compare their reduced forms for syntactic equality to decide whether they are provably equal modulo  $E$ . In particular, the initial algebra  $T_{\Sigma/E}$  associated to a theory  $(\Sigma, E)$  has then a very simple description as the *canonical term algebra*  $\text{Can}_{\Sigma/E}$  whose elements are ground terms that are irreducible by the the equations  $E$ , and whose operations are defined by interpreting  $(\text{Can}_{\Sigma/E})_f(t_1, \dots, t_n)$  as the unique irreducible term obtained by rewriting the term  $f(t_1, \dots, t_n)$  “to the bitter end” with the equations  $E$ . In fact, the existence and initiality of  $\text{Can}_{\Sigma/E}$  can be guaranteed under the weaker assumptions that the equations  $E_B$  are sort-decreasing, ground confluent (confluent for ground terms), and weakly terminating (some rewrite sequence from any term always terminates).

Note that under conditions (ii) and (iii), it is decidable whether confluence (condition (i)) holds, since it is enough to check joinability of the so-called *critical*

*pairs* between the lefthand sides of equations in  $E$  (see, e.g., [10]). Such critical pairs are computed by performing *order-sorted unification* (see, e.g., [39,53,29]) of an equation's lefthand side with a nonvariable subterm of another equation's lefthand side (perhaps for a renamed copy of the same equation).

Since I will make use of order-sorted unifiers in Section 5, precisely in connection with critical pairs, let me give a brief description, along the lines of [29], of how order-sorted unifiers can be represented in terms of their corresponding *unsorted* most-general unifier. It will be useful to view a variable  $x$  as a *name*, and its sort  $s$  as a *sort assignment* for that name in a given context. In this way, the same variable  $x$  can be assigned different sorts in different contexts. If  $\bar{x} = x_1, \dots, x_n$  is a list of variable names, and  $\bar{s} = s_1, \dots, s_n$  is a corresponding list of sort names, I will then write  $\bar{x} : \bar{s}$  for the sort assignment giving sort  $s_i$  to the variable name  $x_i$ . Therefore,  $\bar{x} : \bar{s}$  uniquely specifies the sorts of the variable names  $\bar{x} = x_1, \dots, x_n$ .

Given an order-sorted signature  $\Sigma$  and a  $\Sigma$ -equality  $u = v$ , say with  $\text{vars}(u) \cup \text{vars}(v) = \bar{x} : \bar{s}$ , the order-sorted *unification problem* is then the search for sort-preserving substitutions  $\tau$  such that  $u\tau = v\tau$ . In general there may be none, one, or several most general unifiers. The interesting point, though, is that they all have a simple representation in terms of the single (if it exists) most general *unsorted* unifier  $\sigma = \text{mgu}(u, v)$  obtained when we completely ignore sort information, that is, when we unify  $u$  and  $v$  assuming that their function symbols and their variable names  $\bar{x}$  are all unsorted. Note that we can always assume that such an unsorted most general unifier is of the form  $\sigma = \{x_{i_1} \mapsto t_{i_1}, \dots, x_{i_k} \mapsto t_{i_k}\}$ , with the variables  $x_{i_j}$  and  $\text{vars}(t_{i_{n,j}})$  among the variables  $\bar{x}$ .

Then, each of the most general order-sorted unifiers of  $u = v$  in the context  $\text{vars}(u) \cup \text{vars}(v) = \bar{x} : \bar{s}$  (if any exists) are of the form  $\sigma_{\bar{x} : \bar{s}'}$ , where: (i)  $\bar{s}' = s'_1, \dots, s'_n$  is such that for each  $1 \leq i \leq n$ ,  $s_i \geq s'_i$ , (ii) the unifier  $\sigma_{\bar{x} : \bar{s}'}$  is just the same mapping  $\{x_{i_1} \mapsto t_{i_1}, \dots, x_{i_k} \mapsto t_{i_k}\}$  as the one for  $\sigma$ , but now in the context  $\text{vars}(u) \cup \text{vars}(v) = \bar{x} : \bar{s}'$ , and (iii) for each variable  $x_{i_j}$  we have  $s_{i_j} = s'_{i_j}$ , that is, only the sorts of the variables in the terms  $\text{vars}(t_{i_{n,j}})$  are perhaps lowered in the sort assignment  $\bar{x} : \bar{s}'$ , when compared with those in  $\bar{x} : \bar{s}$ . The sort assignment  $\bar{x} : \bar{s}'$ , is then called a *variable specialization* of the sort assignment  $\bar{x} : \bar{s}$ .

*Example 1.* Let  $\Sigma$  be the signature with sorts  $p, q, r, s$ , ordering  $p \ q < r \ s$ , and operations  $f : r \rightarrow r$ , and  $f : s \rightarrow s$ . Consider the order-sorted unification problem  $f(x) = f(y)$  with sort assignment  $x : r, y : s$ . It has  $\sigma = \text{mgu} = \{x \mapsto y\}$ , and there are two order-sorted unifiers, namely,  $\sigma_{x:r,y:p}$ , and  $\sigma_{x:r,y:q}$ , corresponding to two variable specializations of  $x : r, y : s$ , one where we lower the sort of  $y$  to  $p$ , and another where we lower it to  $q$ .  $\square$

### 3 Pushouts of Signatures and Failure of Exactness

A highly desirable property of a liberal institution [22] is *exactness* [37], which means that the models functor  $\text{Mod} : \text{Sign}^{\text{op}} \longrightarrow \text{Cat}$  maps pushouts of signatures to pullbacks in  $\text{Cat}$ . Exactness, under the assumption that the category

of signatures has pushouts (which ensures that the category of theories also has them), has two important consequences that are very useful for parameterized specifications. Given a theory  $B[P]$  parameterized by  $P$  and a pushout of theories

$$\begin{array}{ccc} B[P] & \xrightarrow{H'} & B[T] \\ J \uparrow & & \uparrow J' \\ P & \xrightarrow{H} & T \end{array}$$

where  $J$  is the inclusion of  $P$  in  $B[P]$ , we: (i) can uniquely describe each  $B[T]$ -model as a pair  $(M, M')$  with  $M$  a  $B[P]$ -model and  $M'$  a  $T$ -model such that<sup>1</sup>  $U_J(M) = U_H(M')$  (the unique  $B[T]$ -model so determined is then called the *amalgamation* of  $M$  and  $M'$ ); and (ii) if  $J$  is *persistent*, that is, if the unit natural morphisms  $\eta_{M_0} : M_0 \longrightarrow U_J F_J(M_0)$  for the adjunction between  $U_J$  and  $F_J$  are isomorphisms, then  $J'$  is also persistent, and, furthermore, for any  $T$ -model  $M'$  its free extension to a  $B[T]$ -model  $F_{J'}(M')$  is precisely the amalgamation of  $M'$  and  $F_J U_H(M')$  (see, e.g., [13] for an account of (i) and (ii)).

Unfortunately, the institution  $OSEql^R$ , although liberal, is not exact, so that we cannot avail ourselves of property (ii) to ensure that persistence is *stable* under pushouts of theories (that is,  $J'$  is persistent if  $J$  is). In general, as I show in Section 4, persistence is *not* stable under pushouts, unless additional properties are satisfied by the theory morphism  $J$ .

Recall that the category **OSSign** of order-sorted signatures and signature morphisms is cocomplete. With small differences (such as the ruling out of ad-hoc overloading in signatures in [50]) this has been well-known since [50]. In applications to parameterized specifications we will only need to consider pushouts of theory inclusions, where such inclusions satisfy a few additional conditions. Therefore, it will be enough for our present purposes to consider pushouts of what I call *guarded* signature inclusions.

**Definition 5.** A signature inclusion  $J : \Sigma \hookrightarrow \Sigma'$  in **OSSign** is a signature morphism such that if  $\Sigma = (\Sigma, (S, \leq))$  and  $\Sigma' = (\Sigma', (S', \leq'))$ , then  $J$  consists of a poset inclusion  $J : S \hookrightarrow S'$  such that for each  $p, q \in S$  we have  $p \leq q$  iff  $p \leq' q$ , and for each  $(w, s) \in S^* \times S$  we have an inclusion  $\Sigma_{w,s} \subseteq \Sigma'_{w,s}$ . The signature inclusion  $J : \Sigma \hookrightarrow \Sigma'$  in **OSSign** is called *guarded* iff: (i)  $\Sigma' - \Sigma$ , defined by  $(\Sigma' - \Sigma)_{w',s'} = \Sigma'_{w',s'} - \Sigma_{w',s'}$  has no ad-hoc overloaded operators, (ii) if  $s \in S$ ,  $s' \in S'$ , and  $s' \leq' s$ , then  $s' \in S$ , and (iii) whenever we have  $f : w \rightarrow s \in \Sigma$  and  $f : w' \rightarrow s' \in \Sigma'$ , with  $w$  and  $w'$  of same length, then  $f : w' \rightarrow s' \in \Sigma$ , that is, subsort-overloaded operators in  $\Sigma$  are never extended by operators with the same name in  $\Sigma'$ .  $J : \Sigma \hookrightarrow \Sigma'$  is called *strongly guarded* iff it is guarded and, in addition, satisfies (iv) any  $f : w' \rightarrow s' \in \Sigma'$  such that  $f : w' \rightarrow s' \notin \Sigma$  is such that  $s' \notin S$ .

---

<sup>1</sup> I use the notation  $U_H$  for the functor  $Mod(H)$ , and  $F_H$  for its left adjoint, which exists because of the liberality assumption.

I summarize below the pushout construction

$$\begin{array}{ccc} \Sigma' & \xrightarrow{H'} & \Sigma' +_{\Sigma} \Sigma'' \\ J \uparrow & & \uparrow J' \\ \Sigma & \xrightarrow{H} & \Sigma'' \end{array}$$

where  $J$  is a guarded signature inclusion, and  $H : \Sigma \rightarrow \Sigma''$  is an arbitrary signature morphism. It turns out that  $J'$  is also a guarded inclusion. To simplify the exposition I will assume without loss of generality that  $S'' \cap (S' - S) = \emptyset$ , and  $\Sigma'' \cap (\Sigma' - \Sigma) = \emptyset$ , where the second intersection refers to just the sets of *names* of the function symbols appearing in the different signatures and not to their diverse typings. If such disjointness conditions are not met, one can easily rename the sorts of  $S' - S$  and the operators of  $\Sigma' - \Sigma$  appearing in the signature  $\Sigma' +_{\Sigma} \Sigma''$  to achieve the same construction. In what follows I will always assume the above disjointness assumptions for notational convenience, in the understanding that when they fail we just need to rename some sorts and some operators in  $\Sigma' +_{\Sigma} \Sigma''$ .

The poset of sorts for  $\Sigma' +_{\Sigma} \Sigma''$  has as set the disjoint union  $(S' - S) \uplus S''$ , and has as its strict ordering the relation

$$<'|_{(S' - S)} \cup <'' \cup \{(s'', s') \mid s'' \in S'' \wedge s' \in (S' - S) \wedge \exists s \in S. s'' \leq'' H(s) \wedge s \leq' s'\}$$

which it is easy to see it is already transitive and irreflexive. It is then also easy to see that this order (adding to it the equality relation) is the pushout, in the category of posets, of the poset morphisms  $J$  and  $H$ , so that we get a monotonic map  $H' : S' \rightarrow (S' - S) \uplus S''$ , and a poset inclusion  $J' : S'' \hookrightarrow (S' - S) \uplus S''$  in the pushout of posets. The signature of  $\Sigma' +_{\Sigma} \Sigma''$  is essentially the disjoint union of the signatures  $\Sigma''$  and  $\Sigma' - \Sigma$ , but where an operator  $f : s'_1 \dots s'_n \rightarrow s'$  in  $\Sigma' - \Sigma$  now has the typing  $f : H'(s'_1) \dots H'(s'_n) \rightarrow H'(s')$ , so that  $H'$  becomes a signature morphism  $H' : \Sigma' \rightarrow \Sigma' +_{\Sigma} \Sigma''$ . It is then easy to see that the obvious inclusion  $J' : S'' \hookrightarrow \Sigma' +_{\Sigma} \Sigma''$  is guarded.

**Proposition 1.** *For any signature morphism  $H : \Sigma \rightarrow \Sigma''$ , the guarded inclusion  $J' : S'' \hookrightarrow \Sigma' +_{\Sigma} \Sigma''$  is the pushout of the guarded inclusion  $J : \Sigma \hookrightarrow \Sigma'$  in the category **OSSign**.*

*Proof.* Consider a pair of signature morphisms  $G' : \Sigma' \rightarrow \Omega$ ,  $G : \Sigma'' \rightarrow \Omega$  such that  $J; G' = H; G$ . We need to show that there is a unique signature morphism  $G'' : \Sigma' +_{\Sigma} \Sigma'' \rightarrow \Omega$  such that  $H'; G'' = G'$ , and  $J'; G'' = G$ . On the poset part  $G''$  is uniquely determined by the poset structure on  $(S' - S) \uplus S''$  being a pushout of posets. The key observation now is that  $(S' - S) \uplus S''$  is also a pushout of sets, and  $\Sigma'' \uplus H'(\Sigma' - \Sigma)$  is a pushout of many-sorted signatures, so there is a unique  $G'' : \Sigma' +_{\Sigma} \Sigma'' \rightarrow \Omega$  such that  $H'; G'' = G'$ , and  $J'; G'' = G$  in the category of *many-sorted* signatures. We only need to check that  $G''$  maps subsort-overloaded families of operators to subsort overloaded families in  $\Omega$ . For the operators in  $\Sigma''$  this follows from  $G$  doing so. For subsort-overloaded operators in  $H'(\Sigma' - \Sigma)$

to be mapped to subsort-overloaded operators in  $\Omega$ , we need to use crucially the assumption that  $\Sigma' - \Sigma$  has no ad-hoc overloaded operators, since otherwise two ad-hoc overloaded operators in  $\Sigma' - \Sigma$  could be mapped by  $G'$  to operators with different names in  $\Omega$ , but could become subsort-overloaded in  $H'(\Sigma' - \Sigma)$ , making it impossible for  $G''$  to exist as a morphism in **OSSign**.  $\square$

We are now ready to see that the institution  $OSEqtl^R$  is not exact.

*Example 2.* (Failure of Exactness). Let  $\Sigma = (\emptyset, (\{s_1, s_2\}, \emptyset))$ , that is,  $\Sigma$  has no operators, and has just two sorts with the discrete (strict) order; and let

$$\Sigma' = (\emptyset, (\{s_1, s_2, s'_1, s'_2\}, \{(s_1, s'1), (s_2, s'2)\}))$$

that is, we just add two new sorts  $s'_1, s'_2$ , with  $s_1 < s'_1$ , and  $s_2 < s'_2$ . The inclusion  $J : \Sigma \hookrightarrow \Sigma'$  is trivially guarded. Let now  $\Sigma'' = (\emptyset, (\{s_0\}, \emptyset))$ , that is, a one-point poset with no operators, and  $H : \Sigma \longrightarrow \Sigma''$  the signature morphism mapping  $s_1$  and  $s_2$  to  $s_0$ . We have  $\Sigma' +_{\Sigma} \Sigma'' = (\emptyset, (\{s_0, s'_1, s'_2\}, \{(s_0, s'_1), (s_0, s'_2)\}))$ . Let  $A$  be the  $\Sigma'$ -algebra given by:  $A_{s_1} = A_{s_2} = \{0\}$ , and  $A_{s'_1} = A_{s'_2} = \mathbb{N}$ , and let  $B$  be the  $\Sigma''$ -algebra given by  $B_{s_0} = \{0\}$ . Obviously  $U_J(A) = U_H(B)$ . But  $\textbf{OSAlg}_{\Sigma' +_{\Sigma} \Sigma''}$  is *not* the pullback of  $U_J$  and  $U_H$  in *Cat*. Indeed, we can define a  $\Sigma' +_{\Sigma} \Sigma''$ -algebra  $A \otimes B$  with  $U_{H'}(A \otimes B) = A$ , and  $U_{J'}(A \otimes B) = B$  (just take  $(A \otimes B)_{s'_1} = (A \otimes B)_{s'_2} = \mathbb{N}$ , and  $(A \otimes B)_{s_0} = \{0\}$ ). But given the identity  $\Sigma''$ -homomorphism  $1_B : B \longrightarrow B$ , and the  $\Sigma'$ -homomorphism  $h : A \longrightarrow A$ , with  $h_{s_1} = h_{s_2} = 1_{\{0\}}$ ,  $h_{s'_1} = 1_{\mathbb{N}}$ , and  $h_{s'_2} = \lambda n. 2 * n$ , which obviously satisfy  $U_J(h) = U_H(1_B)$ , there is no  $\Sigma' +_{\Sigma} \Sigma''$ -homomorphism  $g : A \otimes B \longrightarrow A \otimes B$  such that  $U_{H'}(g) = h$ , and  $U_{J'}(g) = 1_B$ , because  $g$  would have to map 1 to both 1 and 2, which is impossible.  $\square$

## 4 Persistence and Parameterized Convergence Can Be Unstable under Pushouts

In this section I show that important properties of parameterized specifications such as *persistence*, which ensures that “no junk and no confusion” are added to the data in the parameter by its free extension, and *convergence*, which is a basic executability requirement for parameterized specifications as programs, are not stable under pushouts in general. I first define parameterized specifications, the notions of persistence and parameterized convergence, and the instantiation by pushout of a parameterized specification.

**Definition 6.** A parameterized specification is an inclusion of order-sorted theories  $J : P \hookrightarrow B$ , with  $P = (\Sigma_P, E_P)$  called the parameter theory, and  $B = (\Sigma_B, E_B \uplus E_B)$  called the body, such that: (i)  $J$  is guarded; and (ii) we assume that  $E_B$  is a set of unconditional equations of the form  $t = t'$ , with  $t$  not involving any function symbols in  $\Sigma_P$ . To emphasize the parametric nature of  $J : P \hookrightarrow B$ , we will often use the notation  $B[P]$  instead of  $J : P \hookrightarrow B$ , where  $B[P]$  (somewhat ambiguously) can denote both the inclusion  $J$  and the body  $B$ .

A parameterized specification  $B[P]$  is called persistent iff for each  $A \in \mathbf{OSAlg}_P$  the natural  $P$ -homomorphism  $\eta_A : A \longrightarrow U_J F_J(A)$  associated to the adjunction between  $U_J$  and  $F_J$  is an isomorphism.

A parameterized specification  $B[P]$  is called convergent iff the equations  $E_B$  are confluent, terminating, and sort-decreasing.  $\square$

Requirement (ii) implicitly assumes that the equations  $t = t'$  in  $E_B$  will be used as rewrite rules and is quite natural, since  $P$  has “loose” semantics, and therefore we cannot assume anything like constructors for the models of  $P$  when matching with the pattern  $t$ . For the same reason, convergence is formulated exclusively in terms of  $E_B$ : nothing is required of  $E_P$ , since  $E_P$  will not be used at all when  $B[P]$  is instantiated. That is,  $P$  is a *formal requirement* for the instantiations of  $B[P]$ , but  $E_P$  need not have any computational meaning. For example, the transitivity axiom  $x \leq y = \text{true} \wedge y \leq z = \text{true} \Rightarrow x \leq z = \text{true}$ , which may be part the parameter theory of totally ordered sets for a sorting parameterized module, is not executable by rewriting because of the extra variable  $y$ ; but that does not make the sorting equations any less executable. The general idea is that  $P$  has loose semantics, is used only for specification and verification purposes and is totally general; whereas  $B[P]$  has free extension semantics and has equations  $E_B$  that can be used to compute in such free extensions.

A parameterized theory  $B[P]$  can be *instantiated* by means of any theory morphism  $H : P \longrightarrow T$  just by forming the pushout

$$\begin{array}{ccc} B[P] & \xrightarrow{H'} & B[T] \\ J \uparrow & & \uparrow J' \\ P & \xrightarrow{H} & T \end{array}$$

where it would be more precise to denote the pushout theory  $B[T]$  as  $B[H]$ , since one could instantiate  $B[P]$  with different theory morphisms, say,  $H, G : P \longrightarrow T$ . However, I adopt the shorthand notation  $B[T]$  instead of the more awkward but more precise notation  $B[H][T]$  that would unambiguously indicate that  $T$  is the parameter theory of  $B[H]$  obtained by instantiation with  $H$ .

The point is that, since pushouts of theories are obtained by forming the pushout of the underlying signatures and translating the equations, we have  $B[T] = (\Sigma_{B[P]} + \Sigma_T, E_T \oplus H'(E_B))$ , with  $J'$  guarded, and  $B[T]$  a parameterized specification whose parameter theory is  $T$ . This is very useful, since we can construct new parameterized theories out of others by such instantiations.

As a running example, I consider a parameterized specification  $\text{MAP}\{\text{F} :: \text{FUNCTION}\}$  that extends to lists a given function  $\text{f}$  on the basic elements. Here and in other examples I use Maude notation [11], which is self-explanatory. Order-sorted parameter theories are introduced with the keyword `fth...endfth`. Instead, bodies of parameterized theories are introduced with the keyword `fmod...endfm`. The `fth` versus `fmod` keyword distinction makes explicit the *free extension semantics* of the body, so that its intended models are of the form  $F_J(A)$ , for  $A \in \mathbf{OSAlg}_P$ . Theory morphisms used for instantiation are declared with

the `view ... endv` syntax. Here, the obvious theory morphism from `TRIV` to `FUNCTION` is called `FUN`. The `[ctor]` keyword declares that the given operator is a *constructor*. Constructors are further discussed in Sections 6 and 7.

```
fth TRIV is sort Elt . endfth

fmod LIST{X :: TRIV} is sort List .
  subsort X$Elt < List .
  op _;_ : X$Elt List -> List [ctor] .
  op _;_ : List List -> List .
  op first : List -> X$Elt .
  var E : X$Elt . vars L P Q : List .
  eq (L ; P); Q = L ;(P ; Q) .
  eq first(E) = E .
  eq first(E ; L) = E .
endfm

fth FUNCTION is sort Elt .
  op f : Elt -> Elt .
endfth

view FUN from TRIV to FUNCTION is endv

fmod MAP{F :: FUNCTION} is protecting LIST{FUN}{F} .
  op map : List -> List .
  var E : F$Elt . var L : List .
  eq map(E) = f(E) .
  eq map(E ; L) = f(E) ; map(L) .
endfm
```

Let me now focus on failures of persistence and of parameterized convergence under instantiation. Both types of failures are illustrated by the following example, whose parameter theory has two disjoint copies of the `TRIV` theory.

*Example 3.*

```
fmod TRICKY{X :: TRIV, Y :: TRIV} is
  sorts Pair Pair' U .
  suborts Pair Pair' < U .
  op <_,_> : X$Elt X$Elt -> Pair [ctor] .
  op <_,_> : Y$Elt Y$Elt -> Pair' [ctor] .
  op p_ : Pair -> X$Elt .
  op p_ : Pair' -> Y$Elt .
  vars X1 X2 : X$Elt . vars Y1 Y2 : Y$Elt .
  eq p < X1,X2 > = X1 .
  eq p < Y1,Y2 > = Y2 .
endfm

fmod TROUBLE{Z :: TRIV} is
  including TRICKY{Z,Z} .
endfm
```

Note that the equations in TRICKY are trivially sort-decreasing, and clearly terminating (we can associate to  $p$  the polynomial  $x$ , and to  $\langle \_, \_ \rangle$  the polynomial  $y+z$ , so that  $y+z > y$ , and  $y+z > z$ ). Therefore, confluence checking is reduced to checking joinability of critical pairs. Somewhat surprisingly, there are no non-trivial critical pairs. The point is that the sorts  $X\$Elt$  and  $Y\$Elt$  have no subsorts in common, so that order-sorted unification *fails* when trying to unify the terms  $p < X1, X2 >$  and  $p < Y1, Y2 >$ . Therefore, this parameterized specification is convergent. Using the methods in Section 7, it is then easy to show that TRICKY is also persistent. However, the theory morphism from  $TRIV.X \oplus TRIV.Y$  to  $TRIV.Z$  mapping the sorts  $X\$Elt$  and  $Y\$Elt$  to  $Z\$Elt$  gives us the parameterized module  $TROUBLE\{Z :: TRIV\}$  which is *neither convergent nor persistent*. Convergence is lost, because now the variables  $X1$ ,  $X2$ ,  $Y1$  and  $Y2$  all have sort  $Z\$Elt$ , so that  $p < X1, X2 >$  rewrites to *both*  $X1$  and  $X2$ . But this means that the equation  $X1 = X2$  holds in  $TROUBLE\{Z :: TRIV\}$ , utterly destroying persistence, since  $U_JF_J(A)$  must be a one-point set for *any* nonempty set  $A$ .  $\square$

It is easy to show that sort-decreasingness and termination of parameterized specifications can also be lost by instantiation. Therefore, convergence can be lost in three different ways, or by the simultaneous loss of several of these properties.

## 5 Stable Parameterized Convergence

Is it possible to give reasonable conditions under which convergent parameterized specifications are stable under pushouts? This is a very practical question, since it affects the proper operational behavior by rewriting of *parameterized programs* in algebraic languages supporting both order-sortedness and parameterization such as OBJ [26], CafeOBJ [18], Maude [11], and CASL [46].

In this section I give a positive answer to this question by identifying checkable conditions that guarantee stability of both termination and convergence. Preservation of sort-decreasingness for equations in the body is the only extra requirement that is then placed on the theory morphisms used for instantiation of the parameterized specification meeting such conditions. But this is an easy-to-check syntactic condition that should be required anyway.

The counterexample in Section 4 suggests the idea that confluence can be lost by identifying some parameter sorts and/or introducing additional subsorts in the instance  $T$  of the parameter theory  $P$ . Furthermore, such sort identifications and/or new subsorts in the parameter part can allow more rewritings with the translated equations  $H'(E_B)$  than those possible with  $E_B$ , which may also lead to loss of termination.

The following lemma captures these intuitions and is left as an easy exercise.

**Lemma 1.** *Let  $H : T \rightarrow T'$  be a theory morphism, and let  $E_T$  be the equations of  $T$ , which we assume unconditional, sort-decreasing, and such that for each  $t = t'$  in  $E_T$ ,  $\text{vars}(t') \subseteq \text{vars}(t)$ . Assume, furthermore, that the equations  $H(E_T)$  are also sort-decreasing. Then, whenever  $t \rightarrow_{E_T} t'$ , we have  $H(t) \rightarrow_{H(E_T)} H(t')$ . In particular, if  $H(E_T)$  is terminating, then  $E_T$  is terminating too.*  $\square$

The above lemma suggests the idea of looking for a *final object*  $\mathbb{F}$  in the category of order-sorted theories, which we can use to instantiate the parameter  $P$  of a parameterized specification  $B[P]$  in the most brutal and final possible way. Intuitively, if the equations  $E_B$  still terminate under such a final instantiation, they should terminate under *any* instantiation.

Such a final object is not hard to find. First of all, the signature  $\Sigma_{\mathbb{F}}$  with singleton set of sorts  $\{\ast\}$  and with signature  $\mathbb{N}$ , where each  $n \in \mathbb{N}$  is the only operator of  $n$  arguments, that is,  $n : \ast \dots \ast \rightarrow \ast$ , is clearly a final object in the category **OSSign**. And then the theory  $\mathbb{F} = (\Sigma_{\mathbb{F}}, x = 0)$  is a final object in the category of order-sorted equational theories, so that for each theory  $T$  there is a unique theory morphism  $!_T : T \rightarrow \mathbb{F}$ .

**Theorem 4.** *Let  $B[P]$  be a parameterized specification whose equations  $E_B$  are sort-decreasing. If the equations  $!_P(E_B)$  are sort-decreasing and terminating in  $B[\mathbb{F}]$ , then, for any theory morphism  $H : P \rightarrow T$  such that the equations  $H'(E_B)$  are sort-decreasing in  $B[T]$ , they are also terminating. In particular, the equations  $E_B$  are terminating.*

*Proof.* Use Lemma 1, the well-known fact that pushouts of pushouts are pushouts, and the finality of  $\mathbb{F}$ , which gives us the identity  $H; !_T = !_P$ .  $\square$

*Example 4.* We can apply the above theorem to our MAP example to show that all sort-preserving instantiations of the MAP specification are terminating. The equations in  $!_{\text{FUNCTION}}(E_{\text{MAP}})$  are:

```
var E : * . vars L P Q : List .
eq (L ; P) ; Q = L ; (P ; Q) .
eq first(E) = E .
eq first(E ; L) = E .
eq map(E) = 1(E) .
eq map(E ; L) = 1(E) ; map(L) .
```

They are clearly sort-decreasing. They can be proved terminating by the polynomial ordering assigning  $2x + y$  to  $-;$ ,  $2x$  to `first` and `map`, and  $x$  to 1.  $\square$

Can a similar sufficient condition be given to ensure that a confluent parameterized specification will remain confluent after instantiation by a theory morphism making the equations in the body sort-decreasing? The problem is nontrivial because, as we have seen, an instantiation may give rise to new critical pairs between equations that did not arise in the original parameterized specification. Again, a key, easy lemma that sheds light on the problem is:

**Lemma 2.** *Let  $H : \Sigma \rightarrow \Sigma'$  be a signature morphism, and let  $\sigma = \{x : s_1 \mapsto t_1, \dots, x : s_n \mapsto t_n\}$  be an order-sorted unifier of the  $\Sigma$ -equation  $u = v$ . Then,  $H(\sigma) = \{x : H(s_1) \mapsto H(t_1), \dots, x : H(s_n) \mapsto H(t_n)\}$  is a (not necessarily most general) order-sorted unifier of the  $\Sigma'$ -equation  $H(u) = H(v)$ .*  $\square$

The sufficient condition that we seek is given in the following definition, where we use without further comment the representation of order-sorted unifiers discussed in Section 2.

**Definition 7.** Let  $B[P]$  be a convergent parameterized specification, with the equations  $!_P'(E_B)$  sort-decreasing and terminating in  $B[\mathbb{F}]$ .  $B[P]$  is called stably convergent iff for any two equations  $t = t'$  and  $u = u'$  in  $E_B$  (including renamings  $u = u'$  of an equation  $t = t'$ ) in a context  $\bar{x} : \bar{p}, \bar{y} : \bar{q}$ , where  $\bar{p} \in S_P^*$ , and  $\bar{q} \in (S_B - S_P)^*$ , and for each nonvariable position  $p$  in  $t$ , (i) all most general order-sorted unifiers of  $t_p = u$  are of the form  $\sigma_{\bar{x}:\bar{p},\bar{y}:\bar{q}}$ , where only the sorts  $\bar{q} \in (S_B - S_P)^*$  are possibly lowered in the variable specialization  $\bar{x} : \bar{p}, \bar{y} : \bar{q}'$  of  $\bar{x} : \bar{p}, \bar{y} : \bar{q}$  and  $\bar{q}' \in (S_B - S_P)^*$ , and (ii)  $t_p = u$  has unifiers iff  $!_P'(t_p) = !_P'(u)$  has them, and the translations  $!_P'(\sigma_{\bar{x}:\bar{p},\bar{y}:\bar{q}})$ , of such most general unifiers of  $t_p = u$  then provide a complete set of order-sorted unifiers for  $!_P'(t_p) = !_P'(u)$ .  $\square$

*Example 5.* The parameterized module MAP is stably convergent in the above sense. Sort-decreasingness and termination of the equations  $!_{\text{FUNCTION}}(E_{\text{MAP}})$  was checked in Exercise 4. In both  $E_{\text{MAP}}$  and  $!_{\text{FUNCTION}}(E_{\text{MAP}})$ , the only nontrivial (and well-known to be joinable) critical pair is obtained by the overlap of the lefthand side of  $(L ; P) ; Q = L ; (P ; Q)$  with its renamed subterm  $L' ; P'$ , with single most general order-sorted unifier  $\{L' \mapsto L; P, P' \mapsto Q\}$ , in both cases in the context where all variables have sort List, since such a context is unchanging under the translation  $!_{\text{FUNCTION}}(E_{\text{MAP}})$ .  $\square$

That stably convergent parameterized specifications are true to their name is shown by the following theorem.

**Theorem 5.** Let  $B[P]$  be a stably convergent parameterized specification. Then for any theory morphism  $H : P \longrightarrow T$  such that the equations  $H'(E_B)$  are sort-decreasing,  $B[T]$  is convergent.

*Proof.* By Theorem 4, we only need to prove that all nontrivial critical pairs for the equations  $H'(E_B)$  are joinable. Obviously, any such critical pairs will be obtained by means of order-sorted unifiers of equalities  $H'(t_p) = H'(u)$  in a context  $\bar{x} : H(\bar{p}), \bar{y} : \bar{q}$ , where  $t = t'$  and  $u = u'$  are equations in  $E_B$  (including renamings  $u = u'$  of an equation  $t = t'$ ) in a context  $\bar{x} : \bar{p}, \bar{y} : \bar{q}$ . Let  $\sigma_{\bar{x}:\bar{r},\bar{y}:\bar{q}'}$  be one of the most general order-sorted unifiers of  $H'(t_p) = H'(u)$ . Then,  $H(\bar{p}) \geq \bar{r}$ ,  $\bar{q} \geq \bar{q}'$ , and by Lemma 2,  $!_T'(\sigma_{\bar{x}:\bar{r},\bar{y}:\bar{q}'}) = \sigma_{\bar{x}:!_T(\bar{r}),\bar{y}:\bar{q}'}$  is a (not necessarily most general) unifier of  $!_T'H'(t_p) = !_T'H'(u)$ , that is, of  $!_P'(t_p) = !_P'(u)$ , since we have  $!_T; H' = !_P$  by the finality of  $\mathbb{F}$ . Therefore,  $!_P'(t_p) = !_P'(u)$  is unifiable, and therefore  $t_p = u$  is unifiable. This means that there is a most general unifier of  $t_p = u$  in the context  $\bar{x} : \bar{p}, \bar{y} : \bar{q}$  having the form  $\sigma_{\bar{x}:\bar{p},\bar{y}:\bar{q}''}$ , and such that  $!_P'(\sigma_{\bar{x}:\bar{p},\bar{y}:\bar{q}''}) = \sigma_{\bar{x}:!_P(\bar{p}),\bar{y}:\bar{q}''}$  is more general or equal to  $\sigma_{\bar{x}:!_T(\bar{r}),\bar{y}:\bar{q}'}$  by Lemma 2. In particular this means that  $q'' \geq q'$ . But since  $H(\bar{p}) \geq \bar{r}$ , this means that  $H'(\sigma_{\bar{x}:\bar{p},\bar{y}:\bar{q}''}) = \sigma_{\bar{x}:H(\bar{p}),\bar{y}:\bar{q}''}$  is as general or more than our chosen most general unifier  $\sigma_{\bar{x}:\bar{r},\bar{y}:\bar{q}'}$  of  $H'(t_p) = H'(u)$ . This forces  $q'' = q'$ ,  $H(\bar{p}) = \bar{r}$ , and  $H'(\sigma_{\bar{x}:\bar{p},\bar{y}:\bar{q}''}) = \sigma_{\bar{x}:\bar{r},\bar{y}:\bar{q}'}$ . But since the critical pair for the overlap of  $t_p$  and  $u$  with  $\sigma_{\bar{x}:\bar{p},\bar{y}:\bar{q}''}$  is joinable by hypothesis, then by Lemma 1 the critical pair for the overlap of  $H'(t_p)$  and  $H'(u)$  with  $H'(\sigma_{\bar{x}:\bar{p},\bar{y}:\bar{q}''})$  is also joinable, as desired.  $\square$

**Corollary 1.** *Let  $B[P]$  be a stably convergent parameterized specification. Then for any theory morphism  $H : P \rightarrow T$  such that the equations  $H'(E_B)$  are sort-decreasing,  $B[T]$  is stably convergent.*

## 6 Parameterized Free Constructors Are Stably Persistent

The simplest parameterized specifications are parameterized specifications of *free constructors*, where all the data in the body is freely constructed from data in the parameter. Such specifications, since they have  $E_B = \emptyset$ , are stably convergent in a trivial sense. I will show that they are also stably persistent in the following precise sense.

**Definition 8.** *A parameterized specification  $B[P]$  is called stably persistent iff for each theory morphism  $H : P \rightarrow T$ ,  $B[T]$  is persistent. Given a class  $\mathcal{C}$  of theory morphisms containing  $1_P : P \rightarrow P$ ,  $B[P]$  is called stably  $\mathcal{C}$ -persistent iff for each theory morphism  $H : P \rightarrow T$  in  $\mathcal{C}$ ,  $B[T]$  is persistent.*  $\square$

The definition of a parameterized free constructor specification is very simple.

**Definition 9.** *A parameterized theory  $C[P]$  is called a parameterized free constructor specification iff  $J : P \hookrightarrow C$  is strongly guarded and  $E_C = \emptyset$ .*  $\square$

*Example 6.* The subspecification of **LIST** determined by the subsort declaration `subsort X$Elt < List` and the operator declaration

```
op _;-_ : X$Elt List -> List [ctor] .
```

with no equations is a free constructor specification. Similarly, the subspecification of **TRICKY** determined by the two tupling operators is a free constructor specification.  $\square$

The requirement that the constructors are *free* might seem to be somewhat strong, since one can easily imagine constructors that obey some equations. If this is meant just in the sense of obeying some *axioms*, such as commutativity or associativity commutativity, this is just a natural generalization of the above definition to freeness in the modulo case. Rather, what I have in mind is that there may be confluent and terminating equations satisfied by the constructors. In this regard, the key observation is that the freeness requirement is indeed somewhat strong in a *many-sorted* context, but is a quite mild requirement in an *order-sorted* context for two closely-related reasons. On the one hand, we have the possibility of declaring a subsort-overloaded operator to be a constructor on some typings and not in others. This is indeed the case with the `_;-_` operator in the **LIST** example. It then may very well be the case that non constructor terms involving such an operator can be reduced by some equations (for example by the associativity equation in the **LIST** module), while constructor terms are *irreducible*, as it is indeed the case for the above constructor typing for `_;-_` (see Example 8). On the other hand, even if we start with a signature that initially

does not have a rich enough subsort structure to support free constructors, we may use *tree automata techniques*, as done in [5], to refine the subsort structure of our specification into a finer one where constructors now become free for some typings.

Before proving that parameterized free constructor specifications are stably persistent, it will be useful, both for the results in this section and those in Section 7, to recall the free construction  $A \mapsto F_J(A)$  for  $A \in \mathbf{OSAlg}_P$  not just for parameterized free constructor specifications  $C[P]$ , but for any parameterized specification  $B[P]$ . The stable persistence of free constructor specifications  $C[P]$  will then follow as an easy consequence of the general construction  $A \mapsto F_J(A)$  when the parameterized specification is made up of free constructors.

Following ideas in [38], a simple way of describing the  $A \mapsto F_J(A)$  construction is as (the restriction to  $\Sigma_B$  of) an initial algebra construction. Specifically, given  $A \in \mathbf{OSAlg}_P$ ,  $F_J(A)$  is (the restriction to  $\Sigma_B$  of) the initial algebra  $T_{B[\Delta(A)]}$  of the theory  $B[\Delta(A)] = (\Sigma_B(A), E_B \uplus \Delta(A))$ , where  $\Sigma_B(A)$  has the same sort poset as  $\Sigma_B$  and extends  $\Sigma_B$  by adding only the following *new constants*: for each  $s \in S_B$  and each  $a \in A_s$ , we add the new constant  $(a, [s])$  of sort  $s$ . Note that since we can have  $a \in A_s \cap A_{s'}$ , the constant  $(a, [s])$  may have several sorts. However, if  $[s] \neq [s']$ , there will be *different* constants  $(a, [s])$  and  $(a, [s'])$  in the connected components  $[s]$  and  $[s']$  of the poset of sorts. And where  $\Delta(A)$  is the following set of ground equations:

$$\{f((a_1, [s_1]), \dots, (a_n, [s_n])) = (A_f(a_1, \dots, a_n), [s]) \mid f \in \Sigma_{P, s_1 \dots s_n, s}, a_i \in A_{s_i}\}$$

The key point, then, is that if (as I will assume throughout)  $\Sigma_B$  is a sensible signature, then  $\Sigma_B(A)$  is also a sensible signature. Therefore, by Theorem 2, the algebra  $T_{\Sigma_B(A)}$  is an initial  $\Sigma_B(A)$ -algebra, and therefore,  $T_{\Sigma_B(A)/E_B \uplus \Delta(A)}$  is the initial  $B[\Delta(A)]$ -algebra  $T_{B[\Delta(A)]}$ .

**Theorem 6.** *The free construction  $A \mapsto F_J(A)$  for  $A \in \mathbf{OSAlg}_P$  is defined on objects by the defining identity  $F_J(A) = T_{B[\Delta(A)]}|_{\Sigma_B}$ .*

*Proof.* The result follows easily from the following natural bijection of each  $A \in \mathbf{OSAlg}_P$ ,  $D \in \mathbf{OSAlg}_B$ :

$$\mathbf{OSAlg}_P(A, U_J(D)) \cong \mathbf{OSAlg}_B(T_{B[\Delta(A)]}|_{\Sigma_B}, D) : h \mapsto \bar{h}$$

where the key observation is that for each  $D \in \mathbf{OSAlg}_B$ , a  $\Sigma_P$ -homomorphism  $h : A \longrightarrow U_J(D)$  is exactly the same thing as a  $\Sigma_B(A)$ -algebra structure  $(D, h)$  extending the  $\Sigma_B$ -algebra structure  $D$  by the interpretation of each constant  $(a, [s])$  as  $h_s(a)$ , and such that the equations  $\Delta(A)$  are satisfied (since this is what it means to say that  $h$  is a  $\Sigma_P$ -homomorphism). Therefore,  $(D, h)$  is a  $B[\Delta(A)]$ -algebra and there is a unique  $\Sigma_B(A)$ -homomorphism  $\bar{h} : T_{B[\Delta(A)]} \longrightarrow (D, h)$ . The unit of the adjunction is then the natural  $\Sigma_P$ -homomorphism  $\eta_A : A \longrightarrow U_J F_J(A)$  mapping each  $a \in A_s$  to the equivalence class  $[(a, [s])]$  modulo the equations  $E_B \uplus \Delta(A)$ , and we then have the identity  $\eta_A ; U_J(\bar{h}) = h$ .  $\square$

We are almost ready to show that parameterized free constructor specifications are stably persistent. However, before doing so it will be useful to prove a simple technical lemma about the equations  $\Delta(A)$  that will be of help both for parameterized free constructor specifications  $C[P]$  in this section, and for the more general parameterized specifications  $B[P]$  of Section 7.

**Lemma 3.** *For any parameterized specification  $B[P]$  and for any  $A \in \text{OSAlg}_P$  the equations  $\Delta(A)$  are confluent, terminating, and sort-decreasing.*

*Proof.* Since each left-to-right application of an equation

$$f((a_1, [s_1]), \dots, (a_n, [s_n])) = (A_f(a_1, \dots, a_n), [s])$$

to a term  $t$  reduces the number of  $\Sigma_P$ -function symbols by one, the equations  $\Delta(A)$  are clearly terminating. They are also trivially sort-decreasing. Confluence then follows from joinability of nontrivial critical pairs. But there are no such nontrivial critical pairs.  $\square$

We now come to the main theorem of this section.

**Theorem 7.** *Any parameterized free constructor specification  $C[P]$  is stably persistent.*

*Proof.* Since if  $C[P]$  is a free constructor specification then for any  $H : P \rightarrow T$  the specification  $C[T]$  is also a free constructor specification, it is enough to prove that any free constructor specification is persistent. By Lemma 3 this follows easily from the observation that

$$T_{C[\Delta(A)]} = T_{\Sigma_C(A)/\Delta(A)} \cong \text{Can}_{\Sigma_C(A)/\Delta(A)}$$

where  $\text{Can}_{\Sigma_C(A)/\Delta(A)}$  is the *canonical term algebra* (see Section 2) for the confluent, terminating, and sort-decreasing equations  $\Delta(A)$ . But since each constant  $(a, [s])$  is already in  $\Delta(A)$ -canonical form, the unit map  $\eta_A : A \rightarrow U_J(\text{Can}_{\Sigma_C(A)/\Delta(A)})$  maps each  $a \in A_s$  to  $(a, [s])$  is clearly injective. It is also surjective because of the assumption that  $J : P \hookrightarrow C$  is strongly guarded. We still need to show that  $\eta_A$  is an isomorphism, that is, that  $\eta_A^{-1}$  is a homomorphism. This follows from the equalities:

$$\begin{aligned} \eta_A^{-1}((\text{Can}_{\Sigma_C(A)/\Delta(A)})_f((a_1, [s_1]), \dots, (a_n, [s_n]))) &= \eta_A^{-1}(A_f(a_1, \dots, a_n), [s]) = \\ A_f(a_1, \dots, a_n) &= A_f(\eta_A^{-1}(a_1, [s_1]), \dots, \eta_A^{-1}(a_n, [s_n])). \end{aligned} \quad \square$$

## 7 Sufficient Completeness and Stable Persistence

The same way that constructors play a key role in both checking the definedness of functions in initial algebras and proving inductive theorems about them, *parameterized free constructor specifications* play a completely similar key role for parameterized specifications. In particular, since in general operators from the

body may inject data back into parameter sorts, constructors are very useful to ensure that a parameterized specification is persistent. For this, the key notion to check is the *sufficient completeness* of  $B[P]$  with respect to a subspecification  $C[P]$  of free constructors. As we shall see, under quite general conditions, stably convergent parameterized specifications that are sufficiently complete are also stably persistent for all instantiations making the equations in their body sort-decreasing.

**Definition 10.** Let  $B[P]$  be a convergent parameterized specification. A subspecification of parameterized free constructors for  $B[P]$  is a free constructor specification  $C[P]$  with same parameter  $P$  and same poset of sorts such that there is a subtheory inclusion  $K : (\Sigma_C, EP \sqcup \emptyset) \hookrightarrow (\Sigma_B, EP \sqcup EB)$ .  $\square$

Example 6 already discussed examples of free constructor subspecifications for the parameterized specifications LIST and TRICKY.

**Definition 11.** A convergent parameterized specification  $B[P]$  is called sufficiently complete with respect to a subspecification  $C[P]$  of free constructors iff for each  $t \in T_{\Sigma_B}(X_P)$ , where  $X_P = \{X_s \mid s \in S_P\}$  is a disjoint family of countable sets of variables for each parameter sort  $s \in S_P$ , there is a term  $t_0 \in T_{\Sigma_C}(X_P)$  such that  $t \rightarrow_{EB}^* t_0$ .  $\square$

Intuitively, sufficient completeness tells us that all data for terms in the body can be built up by constructors from data in the parameter, so that any function defined in the body ultimately evaluates to such constructor data. Of course, we would like to check sufficient completeness *once and for all*, and be sure that it holds not just for  $B[P]$ , but for any instance  $B[T]$ . That is, we would like sufficient completeness to be a property *stable under pushouts*.

**Theorem 8.** Let  $B[P]$  be a stably convergent parameterized specification that is sufficiently complete with respect to a free constructor subspecification  $C[P]$ . Then for any theory morphism  $H : P \longrightarrow T$  such that the equations  $H'(EB)$  are sort-decreasing,  $B[T]$  is sufficiently complete with respect to the free constructor subspecification  $C[T]$ .

*Proof.* Let  $H : P \longrightarrow T$  be such that the equations  $H'(EB)$  are sort-decreasing, and suppose that there is a term  $t \in T_{\Sigma_B}(X_T)$  for which we cannot find a term  $t_0 \in T_{\Sigma_C}(X_T)$  such that  $t \rightarrow_{H'(EB)}^* t_0$ . Of course since by Theorem 5  $B[T]$  is convergent, we may assume without loss of generality that such a  $t$  is in canonical form by the equations  $H'(EB)$ . Furthermore, we can choose  $t$  of *minimal depth* as a term satisfying such a property. This means that  $t$  is of the form  $f(t_1, \dots, t_n)$  and has a minimal sort  $H(s)$  associated to an operator  $f : s_1 \dots s_n \longrightarrow s \in \Sigma_B - \Sigma_C$  with each  $t_i \in T_{\Sigma_C}(X_T)$  of sort  $H(s_i)$ . The theorem will then follow if we can prove that for each  $u \in T_{\Sigma_C}(X_T)$  of sort  $H(s)$  there is a linear term  $\tilde{u} \in T_{\Sigma_C}(X_P)$  of sort  $s$  and a substitution  $\sigma$  such that  $u = H'(\tilde{u})\sigma$ , since then we will be able to build a term  $f(\tilde{t}_1, \dots, \tilde{t}_n) \in T_{\Sigma_C}(X_P)$  such that there is a substitution  $\sigma$  such that  $f(t_1, \dots, t_n) = f(\tilde{t}_1, \dots, \tilde{t}_n)\sigma$ . But

this means that  $f(\tilde{t}_1, \dots, \tilde{t}_n)$  must have minimal sort  $s$  and not be a constructor term. Therefore,  $f(\tilde{t}_1, \dots, \tilde{t}_n)$  is reducible by  $E_B$ , and a fortiori, by Lemma 1,  $f(t_1, \dots, t_n)$  is reducible by  $H'(E_B)$ , a blatant contradiction. Therefore, all boils down to proving:

**Lemma 4.** *For each  $u \in T_{\Sigma_C}(X_T)$  of sort  $H(s)$  there is a linear term  $\tilde{u} \in T_{\Sigma_C}(X_P)$  of sort  $s$  and a substitution  $\sigma$  such that  $u = H'(\tilde{u})\sigma$ .*

*Proof.* Suppose that the lemma fails. We then have a term  $u \in T_{\Sigma_C}(X_T)$  of sort  $H(s)$  and of *minimal depth* such that we cannot find a linear term  $\tilde{u} \in T_{\Sigma_C}(X_P)$  of sort  $s$  and a substitution  $\sigma$  such that  $u = H'(\tilde{u})\sigma$ . Obviously, the sort of  $u$  must belong to  $S_B - S_T$ , since otherwise we could pick for our  $\tilde{u}$  a variable in  $X_P$ . Therefore, there is a constructor operator  $c : s'_1 \dots s'_m \longrightarrow s' \in (\Sigma_C - \Sigma_P)$  such that  $s' \leq s$  and terms  $v_i \in T_{\Sigma_C}(X_T)$  of sort  $H(s'_i)$  such that  $u = c(v_1, \dots, v_m)$ . But since  $u$  is of minimal depth failing the desired property, there are also linear terms  $\tilde{v}_i \in T_{\Sigma_C}(X_P)$  (which we may safely assume have  $\text{vars}(\tilde{v}_i) \cap \text{vars}(\tilde{v}_j) = \emptyset$  when  $i \neq j$ ) and substitutions  $\sigma_i$  such that  $v_i = H'(\tilde{v}_i)\sigma_i$ . But then we have a linear constructor term  $\tilde{u} = c(\tilde{v}_1, \dots, \tilde{v}_m)$  and a substitution  $\sigma = \sigma_1 \uplus \dots \uplus \sigma_m$  such that  $u = c(v_1, \dots, v_m) = H'(c(\tilde{v}_1, \dots, \tilde{v}_m))\sigma$ , a blatant contradiction.  $\square$

The relationship between sufficient completeness and persistence is clarified by the following theorem.

**Theorem 9.** *Let  $B[P]$  be a convergent parameterized specification that is sufficiently complete with respect to a free constructor subspecification  $C[P]$  and such that: (i) the equations  $E_B$  are left-linear,<sup>2</sup> and (ii) any constructor term  $t_0 \in T_{\Sigma_C}(X_P)$  is in  $E_B$ -canonical form (the constructors are really free in  $B[P]$ ). Then  $B[P]$  is persistent.*

*Proof.* We need to show that for each  $A \in \mathbf{OSAlg}_P$  the natural  $P$ -homomorphism  $\eta_A : A \longrightarrow U_J F_J(A)$  is an isomorphism. We first need some technical results.

**Lemma 5.** *For  $B[T]$  with  $E_T$  sort-decreasing and left-linear, let  $t \in T_{\Sigma_B}(A)$  be such that  $t \rightarrow_{E_B} t'$  and  $t \rightarrow_{\Delta(A)} u$ . Then there exists a term  $u'$  such that  $u \rightarrow_{E_B} u'$  and  $t' \rightarrow_{\Delta(A)}^* u'$ .*

*Proof.* Since the rules in  $\Delta(A)$  are ground and do not share any symbols with the lefthand sides of the rules in  $E_B$ , if  $p$  is the position at which the rewrite  $t \rightarrow_{E_B} t'$  happens, and  $q$  the position at which the rewrite  $t \rightarrow_{\Delta(A)} u$  happens, then either  $p$  and  $q$  are disjoint positions and the lemma follows trivially, or  $q$  must happen within a subterm of the form  $\sigma(x)$  for  $x$  one of the variables of  $l$  in the equation  $l = r$  used for the rewrite  $t \rightarrow_{E_B} t'$ , with  $t = t[l\sigma]_p$ , and  $t' = t[r\sigma]_p$ . But since  $l$  is left-linear, and the rules in  $\Delta(A)$  are sort-decreasing, this means that we can rewrite  $u$  at the same position  $p$  with the same rule  $l = r$  to obtain a term  $u'$  which only differs from  $t'$  in that subterms of the form  $\sigma(x)$  below  $p$

---

<sup>2</sup> This just means that they are of the form  $t = t'$ , with each variable occurring *exactly once* in  $t$ .

have been rewritten with  $\Delta(A)$ . Let  $n$  be the number of occurrences of  $x$  in  $r$ . Then we need exactly  $n$  applications of the same rule of  $\Delta(A)$  used in  $t \rightarrow_{\Delta(A)}^* u$  to each of the  $n$  copies of the term  $\sigma(x)$  in  $t' = t[r\sigma]_p$  to reach  $u'$ .  $\square$

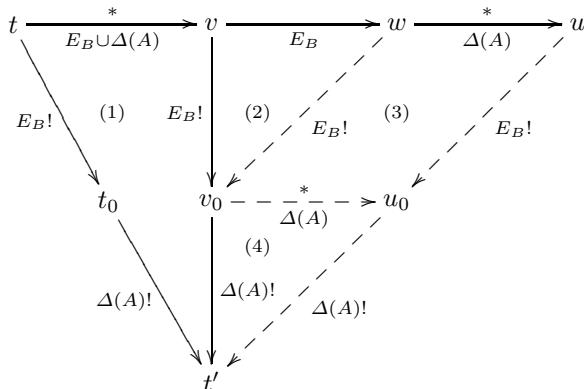
**Corollary 2.** *For  $B[T]$  with  $E_T$  sort-decreasing and left-linear, let  $t \in T_{\Sigma_B(A)}$  be such that  $t \rightarrow_{E_B}^* t'$  and  $t \rightarrow_{\Delta(A)}^* u$ . Then there exists a term  $u'$  such that  $u \rightarrow_{E_B}^* u'$  and  $t' \rightarrow_{\Delta(A)}^* u'$ .*

**Proposition 2.** *Let  $B[P]$  satisfy the conditions in Theorem 9. Then the equations  $E_B \uplus \Delta(A)$  are weakly terminating and confluent for all terms  $t \in T_{\Sigma_B(A)}$ .*

*Proof.* Weak termination follows from the observation that any term  $t \in T_{\Sigma_B(A)}$  can be expressed as a term of the form  $t = \hat{t}\sigma$  with  $\hat{t} \in T_{\Sigma_B}(X_P)$  and  $\sigma$  mapping each variable  $x : s$  to a constant  $(a, [s])$  such that  $a \in A_s$ . By sufficient completeness we then have a rewrite sequence  $\hat{t} \rightarrow_{E_B} \hat{t}_0$  with  $\hat{t}_0 \in T_{\Sigma_C}(X_P)$ , and therefore a rewrite sequence  $t = \hat{t}\sigma \rightarrow_{E_B} \hat{t}_0\sigma$ , where, by the assumptions about  $B[P]$ , both  $\hat{t}_0$  and  $t_0 = \hat{t}_0\sigma$  are in  $E_B$ -canonical form. But then we can use Lemma 3 to further rewrite  $t_0 = \hat{t}_0\sigma$  with the equations  $\Delta(A)$  to a term  $t'$  irreducible by  $E_B \uplus \Delta(A)$ , as desired. Note that, by the respective confluence of  $E_B$  and  $\Delta(A)$ , the terms  $t_0$  and  $t'$  such that<sup>3</sup>  $t \rightarrow_{E_B}^* t_0 \rightarrow_{\Delta(A)}^* t'$  are uniquely determined by  $t$ . Confluence now follows trivially from the following lemma.

**Lemma 6.** *For any  $t \in T_{\Sigma_B(A)}$  and any rewrite sequence  $t \rightarrow_{E_B \uplus \Delta(A)}^* u$ , the rewrite sequences  $t \rightarrow_{E_B}^* t_0 \rightarrow_{\Delta(A)}^* t'$  and  $u \rightarrow_{E_B}^* u_0 \rightarrow_{\Delta(A)}^* u'$  are such that  $t' = u'$ .*

*Proof.* We reason by induction on the number  $n$  of  $E_B$ -rewrites in the rewrite sequence  $t \rightarrow_{E_B \uplus \Delta(A)}^* u$ . If  $n = 0$ , the results follows easily from Corollary 2 using the confluence of  $\Delta(A)$  shown in Lemma 3. Assuming the result for sequences involving  $n$   $E_B$ -rewrites, we consider a rewrite sequence of the form  $t \rightarrow_{E_B \uplus \Delta(A)}^* v \rightarrow_{E_B} w \rightarrow_{\Delta(A)}^* u$ , where  $t \rightarrow_{E_B \uplus \Delta(A)}^* v$  involves  $n$   $E_B$ -rewrites. The proof then is summarized in the diagram



<sup>3</sup> By convention,  $t \rightarrow_E^! t'$  means that  $t \rightarrow_E^* t'$  and  $t'$  is  $E$ -irreducible.

where (1) holds by the induction hypothesis, (2) by confluence of  $E_B$ , (3) by Corollary 2, and (4) by the confluence of  $\Delta(A)$ .  $\square$

Since the equations  $E_B \uplus \Delta(A)$  are also sort-decreasing, their ground confluence and weak termination ensure that we have a very simple description of  $T_{B[\Delta(A)]}$  as the *canonical term algebra* (see Section 2)  $\text{Can}_{\Sigma_B(A)/E_B \uplus \Delta(A)}$ . But since by the proof of Proposition 2 each  $t \in \text{Can}_{\Sigma_B(A)/E_B \uplus \Delta(A)}$  is a constructor term we have an *identity* of canonical term algebras

$$\text{Can}_{\Sigma_C(A)/\Delta(A)}|_{\Sigma_C} = \text{Can}_{\Sigma_B(A)/E_B \uplus \Delta(A)}|_{\Sigma_C}$$

which is of course also an isomorphism of  $\Sigma_C$ -algebras. Also, by Theorem 7 we have a  $P$ -isomorphism  $\eta'_A : A \longrightarrow \text{Can}_{\Sigma_C(A)/\Delta(A)}|_{\Sigma_P}$ . Then, using a decomposition of  $J : P \hookrightarrow B$  as

$$P \xrightarrow{J'} C \xrightarrow{K} B$$

together with the identities  $U_J = U_K; U_{J'}$ , and  $F_J = F_{J'}; F_K$ , we obtain that the identity isomorphism  $\text{Can}_{\Sigma_C(A)/\Delta(A)}|_{\Sigma_C} = \text{Can}_{\Sigma_B(A)/E_B \uplus \Delta(A)}|_{\Sigma_C}$  is the unit map for the adjunction between  $U_K$  and  $F_K$  for the  $C[P]$ -algebra  $F_{J'}(A)$ . We are now essentially done, since by the well-known yoga about composition of adjunctions (see, e.g., [36]), if  $\eta'_A : A \longrightarrow U_{J'}F_{J'}(A)$  is the unit for the adjunction between  $U_{J'}$  and  $F_{J'}$ , and  $\eta''_C : C \longrightarrow U_KF_K(C)$  is the unit for the adjunction between  $U_K$  and  $F_K$ , then the unit  $\eta_A : A \longrightarrow U_JF_J(A)$  for the adjunction between  $U_J$  and  $F_J$ , is precisely the composed map

$$A \xrightarrow{\eta'_A} F_{J'}(A) \xrightarrow{U_{J'}(\eta''_{F_{J'}(A)})} U_{J'}U_KF_KF_{J'}(A)$$

But since  $\eta''_{F_{J'}(A)} : \text{Can}_{\Sigma_C(A)/\Delta(A)}|_{\Sigma_C} \longrightarrow \text{Can}_{\Sigma_B(A)/E_B \uplus \Delta(A)}|_{\Sigma_C}$  is precisely the *identity*  $\Sigma_P$  isomorphism  $1_{F_{J'}(A)}$ , we therefore get that  $\eta_A : A \longrightarrow U_JF_J(A)$  is *identical* with the  $P$ -isomorphism  $\eta'_A : A \longrightarrow \text{Can}_{\Sigma_C(A)/\Delta(A)}|_{\Sigma_P}$ , and is therefore a  $P$ -isomorphism, as desired.  $\square$

We are now ready for the main result of this section about the stable persistence of sufficiently complete parameterized specifications.

**Theorem 10.** *Let  $B[P]$  be stably convergent and sufficiently complete with respect to a free constructor subspecification  $C[P]$ , and such that the equations  $E_B$  are left-linear and each term  $t_0 \in T_{\Sigma_{C[\mathbb{F}]}(X_{\mathbb{F}})}$  is in  $!_P(E_B)$ -canonical form. Then  $B[P]$  is stably SD-persistent, where SD is the class of all theory morphism  $H : P \longrightarrow T$  such that the equations  $H'(E_B)$  are sort-decreasing.*

*Proof.* By Theorem 5,  $B[T]$  is convergent, and by Theorem 8 it is sufficiently complete. Furthermore, by the assumption that terms  $t_0 \in T_{\Sigma_{C[\mathbb{F}]}(X_{\mathbb{F}})}$  are in  $!_P(E_B)$ -canonical form and the finality of  $\mathbb{F}$  we must have each  $t_0 \in T_{\Sigma_{C[T]}}(X_T)$  is in  $H'(E_B)$ -canonical form, since otherwise, by Lemma 1,  $!_T(t_0)$  would not be in  $!_P(E_B)$ -canonical form. Therefore,  $B[T]$  is persistent by Theorem 9.  $\square$

**Corollary 3.** Let  $B[P]$  be stably convergent and sufficiently complete with respect to a free constructor subspecification  $C[P]$ , and such that the equations  $E_B$  are left-linear and each term  $t_0 \in T_{\Sigma_{C[P]}}(X_{\mathbb{F}})$  is in  $!_P(E_B)$ -canonical form. Then for each  $H : P \rightarrow T$  such that the equations  $H'(E_B)$  are sort-decreasing the parameterized module  $B[T]$  is stably SD-persistent.

One practical question to consider is how the conditions for  $B[P]$  required in Theorem 10 can be checked in practice. The checking of conditions for stable convergence has already been discussed and illustrated with examples in Section 5, so the main question is how to check sufficient completeness and the requirement that the terms  $t_0 \in T_{\Sigma_{C[\mathbb{F}]}}(X_{\mathbb{F}})$  are in  $!_P(E_B)$ -canonical form.

For sufficient completeness we can adapt to the parameterized case well-known tree automata techniques for checking the sufficient completeness of left-linear initial algebra specifications (see, e.g., [12,28]). The key observation is that term rewriting with  $E_B$  is performed in the exact same way in the term algebra  $T_{\Sigma_B}(X_P)$  and in the initial algebra  $T_{\Sigma_B(X_P)}$ , where the variables in  $X_P$  have been added to the signature as constants. The problem is that  $\Sigma_{B(X_P)}$  is an *infinite* signature. However, if we can obtain a suitable *finite* signature to represent the initial algebra  $T_{\Sigma_B(X_P)}$ , we should be able to use standard tree automata techniques to check sufficient completeness in the parameterized sense.

We can generate the countable sets of variables  $X_q$  for each parameter sort  $q \in S_P$  by adding to  $\Sigma_B$  a new sort *NewNat* with a zero constant 0 and a successor function  $s : \text{NewNat} \rightarrow \text{NewNat}$ , and for each parameter sort  $q \in S_P$  a new “variable generating” operation  $_n : q : \text{NewNat} \rightarrow q$ , so that we can represent the  $n$ th variable of sort  $q$ , say,  $x_n : q$ , by the term  $n : q$ .

In this way we obtain a *finite* signature  $\Sigma_{\widehat{B}}$  such that the initial algebra  $T_{\Sigma_{\widehat{B}}}$ , when restricted to  $\Sigma_B$ , is *isomorphic* to the free term algebra  $T_{\Sigma_B}(X_P)$ , that is, the mapping  $h : x_n : q \mapsto n : q$  extends to a  $\Sigma_B$ -isomorphism  $\bar{h} : T_{\Sigma_B}(X_P) \rightarrow T_{\Sigma_{\widehat{B}}}|_{\Sigma_B}$ , so that we have  $t \rightarrow_{E_B} t'$  iff  $\bar{h}(t) \rightarrow_{E_B} \bar{h}(t')$ . Then, checking sufficient completeness of  $B[P]$  with respect to  $C[P]$  can be reduced to checking sufficient completeness in the *standard*, initial algebra sense for the initial algebra associated to the specification  $(\Sigma_{\widehat{B}}, E_B)$  with respect to the *standard* constructor subsignature made up by: (i) the subsignature  $\Sigma_C$ , and (ii) the subsignature  $\Sigma_{\widehat{B}} - \Sigma_B$ .

*Example 7.* Let me illustrate this method by checking the sufficient completeness of the parameterized specification **MAP**. In this case, the specification  $(\Sigma_{\widehat{B}}, E_B)$  and its subsignature of constructors can be specified as shown below. The output of Maude’s tree-automata-based Sufficient Completeness Checker [30] is also included.

```
fmod MAP-SUFF-COMP is
  sorts Elt List NewNat .  subsorts Elt < List .

  *** constructors
  op f : Elt -> Elt [ctor] .
  op _:_ : Elt List -> List [ctor] .
```

```

op 0 : -> NewNat [ctor] .
op s : NewNat -> NewNat [ctor] .
op _:Elt : NewNat -> Elt [ctor] .

*** defined functions
op _:_ : List List -> List .
op first : List -> Elt .
op map : List -> List .

*** equations
var E : Elt . vars L P Q : List .
eq (L ; P); Q = L ;(P ; Q) .
eq first(E) = E .
eq first(E ; L) = E .
eq map(E) = f(E) .
eq map(E ; L) = f(E) ; map(L) .
endfm

```

Maude> (scc MAP-SUFF-COMP .)

Success: MAP-SUFF-COMP is sufficiently complete assuming  
it is ground weakly-normalizing, confluent, and  
ground sort-decreasing.

The second issue to consider is how to check that all the terms  $t_0 \in T_{\Sigma_C[\mathbb{F}]}(X_{\mathbb{F}})$  are in  $!'_P(E_B)$ -canonical form. Since the equations  $!'_P(E_B)$  are convergent, we can use a narrowing analysis for this purpose. The point is that if there is a constructor term that is not in canonical form, then there is a one step rewrite with the rules  $!'_P(E_B)$  at the *top* of a constructor term  $c(t_1, \dots, t_n)$ , with, say,  $c : s_1 \dots s_n \longrightarrow s$ , and with the  $t_1, \dots, t_n$  irreducible. Therefore, by the completeness of narrowing [32], there is a narrowing step  $c(x_1 :!'_P(s_1), \dots, x_n :!'_P(s_n)) \rightsquigarrow t$  for some  $t$ . As a consequence, to check that all the terms  $t_0 \in T_{\Sigma_C[\mathbb{F}]}(X_{\mathbb{F}})$  are in  $!'_P(E_B)$ -canonical form it is enough to show that the terms  $c(x_1 :!'_P(s_1), \dots, x_n :!'_P(s_n))$  for each  $c : s_1 \dots s_n \longrightarrow s$  in  $\Sigma_C$  cannot be narrowed with the equations  $!'_P(E_B)$ .

*Example 8.* Let me illustrate the checking of this second property with the MAP specification. The equations  $!_{\text{FUNCTION}}(E_{\text{MAP}})$  were already described in detail in Example 4. We now have to show that neither the term  $E ; L$  nor the term  $1(E)$ , where  $E$  has sort  $*$ , and  $L$  has sort  $\text{List}$ , can be narrowed with the equations  $!_{\text{FUNCTION}}(E_{\text{MAP}})$ . We can use the Full Maude narrowing search command to check these facts. The resulting output is as follows:

Maude> (search in MAP-CTOR-FREE : E:/\* ; L:List ~>1 Q:List .)

No more solutions.

Maude> (search in MAP-CTOR-FREE : 1(E:/\*) ~>1 X:/\* .)

No more solutions.

## 8 Parameterized Induction

What theorems are true for all the free extensions  $F_J(A)$  of all  $P$ -algebras  $A \in \mathbf{OSAlg}_P$ ? And what reasoning principles are sound to infer such theorems? Since freeness generalizes induction, such theorems should be thought of as *parameterized inductive theorems*, and the reasoning principles as *inductive* reasoning principles.

First of all we need to make clear what sentences we are going to use to express such inductive theorems. The obvious choice is *order-sorted first-order logic with equality*. Given an order-sorted signature  $\Sigma$ , let us denote the set of its first-order sentences by  $OSFOL(\Sigma)$ . Such sentences are *fully quantified formulas* built up in the usual way, that is:

1. atomic formulas are  $\Sigma$ -equations  $t = t'$
2. if  $\varphi, \psi$  are formulas, then  $\neg\varphi$ ,  $\varphi \vee \psi$ , and  $\varphi \wedge \psi$  are formulas, and
3. if  $\varphi$  is a formula and  $x : s$  is a variable of sort  $s \in S$ , then  $\forall x : s. \varphi$ , and  $\exists x : s. \varphi$  are also formulas.

The class of *models*  $A$  on which the satisfaction  $A \models \varphi$  of a sentence  $\varphi \in OSFOL(\Sigma)$  is defined in the standard way is of course  $\mathbf{OSAlg}_\Sigma$ . A *first-order*  $\Sigma$ -theory is then a pair  $(\Sigma, \Gamma)$  with  $\Gamma \subseteq OSFOL(\Sigma)$ . We then define the category of models of the theory  $(\Sigma, \Gamma)$  in the obvious way, as the full subcategory of  $\mathbf{OSAlg}_\Sigma$  determined by the class of models

$$\mathbf{OSAlg}_{(\Sigma, \Gamma)} = \{A \in \mathbf{OSAlg}_\Sigma \mid \forall \varphi \in \Gamma, A \models \varphi\}$$

As usual, given a class of algebras  $\mathcal{C} \subseteq \mathbf{OSAlg}_\Sigma$ , and a set of sentences  $\Gamma \subseteq OSFOL(\Sigma)$ , we write  $\mathcal{C} \models \Gamma$  iff  $\forall A \in \mathbf{OSAlg}_\Sigma, \forall \varphi \in \Gamma, A \models \varphi$ . Likewise, given  $\Gamma, \Gamma' \subseteq OSFOL(\Sigma)$ , we write  $\Gamma \models \Gamma'$  iff  $\mathbf{OSAlg}_{(\Sigma, \Gamma)} \models \Gamma'$ . Of course, unconditional (resp. conditional) equational theories  $(\Sigma, E)$  are just special cases of first-order theories, namely, those whose sentences are universally quantified equations (resp. universally quantified Horn clauses).

**Definition 12.** (*Parameterized Inductive Satisfaction and Theoremhood*). Let  $B[P]$  be a parameterized equational theory. Its inductive satisfaction relation, denoted  $B[P] \models_{ind} \varphi$ , is defined for each  $\varphi \in OSFOL(\Sigma_B)$  by the defining equivalence

$$B[P] \models_{ind} \varphi \Leftrightarrow \mathbf{FOSAlg}(B[P]) \models \varphi$$

where, by definition,  $\mathbf{FOSAlg}(B[P]) = \{F_J(A) \mid A \in \mathbf{OSAlg}_P\}$ . Since  $F_J(A)$  is only determined up to isomorphism, we regard the class  $\mathbf{FOSAlg}(B[P])$  as closed by isomorphisms. The set  $IndThm(B[P])$  of the inductive theorems of  $B[P]$  is, by definition, the set of sentences

$$IndThm(B[P]) = \{\varphi \in OSFOL(\Sigma_B) \mid B[P] \models_{ind} \varphi\}. \quad \square$$

Since an unparameterized equational theory  $B$  is the special case  $B[\emptyset]$  of a parameterized theory whose parameter is the empty theory with empty set of sorts

and empty set of sentences, so that the initial algebra  $T_B$  is the free extension  $F_J(\emptyset)$  of the only, empty model of the theory  $\emptyset$ , the unparameterized inductive satisfaction relation  $B \models_{ind}$  defined by the equivalence  $B \models_{ind} \varphi \Leftrightarrow T_B \models \varphi$  is the special case  $B[\emptyset] \models_{ind}$  of the parameterized inductive satisfaction relation  $B[P] \models_{ind}$ . Therefore, by Gödel's incompleteness theorem there is no hope in general of obtaining a *complete* inference system  $B[P] \vdash_{ind}$  to derive all the parameterized inductive theorems of  $B[P]$ .

Nevertheless, one can come up with *sound* inference systems  $B[P] \vdash_{ind}$  which can be used in practice to derive many parameterized inductive theorems. For example, “inductionless induction” types of parameterized inductive inference systems have been proposed for many-sorted parameterized specifications, in, e.g., [34,4], and for parameterized membership equational logic specifications in [6]. I give below a simple parameterized inductive inference system based on *order-sorted structural induction* which is well-suited for interactive inductive proofs in the style of the Maude ITP [9,10,31]. It is of course one possible choice of inference systems among many, and (as I explain below) it is not the only choice even for structural induction. But it should be capable of proving useful theorems in practice. I rely on a sound and complete inference system  $\vdash_{OSFOL}$  for order-sorted first-order logic. Of course, for order-sorted Horn logic one can rely on the sound and complete inference subsystem presented in [38].

**Definition 13.** (*Parameterized Inductive Inference*). Let  $B[P]$  be a convergent parameterized specification that is sufficiently complete with respect to a subspecification  $C[P]$  of free constructors and such that: (i) the equations  $E_B$  are left-linear, and (ii) any constructor term  $t_0 \in T_{\Sigma_C}(X_P)$  is in  $E_B$ -canonical form. The inference system  $B[P] \vdash_{ind}$  is specified for each  $\varphi \in OSFOL(\Sigma_B)$  by the defining equivalence

$$B[P] \vdash_{ind} \varphi \Leftrightarrow E_B \cup E_P \cup SIndSch(B[P]) \vdash_{OSFOL} \varphi$$

where  $SIndSch(B[P])$ , the scheme of parameterized structural induction, is defined as the set<sup>4</sup>

$$SIndSch(B[P]) = \{SInd(\forall x:s. \varphi) \mid x:s \in X_s, s \in (S_B - S_P), \forall x:s. \varphi \in OSFOL(\Sigma_B)\}$$

where each  $X_s$  is a countable set of variables, and where  $SInd(\forall x:s. \varphi)$  is the sentence:

$$[(\bigwedge_{s' \prec s} \forall x:s'. \varphi(x:s')) \wedge (\bigwedge_{c \in \Sigma_C, s_1, \dots, s_n, s} ((\bigwedge_{s_i \leq s} \varphi(x_i:s_i)) \Rightarrow \varphi(c(x_1:s_1, \dots, x_n:s_n))))]$$

$$\Rightarrow \forall x:s. \varphi$$

---

<sup>4</sup> To simplify notation in what follows, when I write  $\forall x:s. \varphi$  I do not assume that  $x:s$  is syntactically the *last* variable that has been universally quantified. Instead, I assume that the formula has the form  $\forall x_1:s_1, \dots, x:s, \dots x_n:s_n. \varphi$ , with the usual notational extension that allows several variables to be universally quantified at the same time. Similarly, an expression such as  $\varphi(c(x_1:s_1, \dots, x_n:s_n))$  is shorthand notation for the substitution instance  $\varphi(x:s \mapsto c(x_1:s_1, \dots, x_n:s_n))$ .

where  $s' \prec s$  denotes the immediate subsort relation,<sup>5</sup> that is,  $s' < s$  and there is no  $s''$  such that  $s' < s'' < s$ , and where the variables  $x:s'$  and  $x_1:s_1, \dots, x_n:s_n$  are assumed fresh.  $\square$

*Example 9.* Let me illustrate the use of the scheme of parameterized structural induction by proving the parameterized inductive theorem

$$\text{MAP}\{\text{F} :: \text{FUNCTION}\} \vdash_{\text{ind}} \forall L, L' : \text{List}. \text{ map}(L ; L') = \text{map}(L) ; \text{map}(L')$$

We can induct on the variable  $L$ . Since the sort  $\text{Elt}$  is the only immediate subsort of  $\text{List}$ , and the only constructor is  $\_ ; \_ : \text{Elt List} \longrightarrow \text{List}$ , the structural induction scheme applied to our sentence reduces the proof to proving the two sentences:

$$\forall E : \text{Elt}, \forall L' : \text{List}. \text{ map}(E ; L') = \text{map}(E) ; \text{map}(L')$$

and

$$\begin{aligned} \forall E : \text{Elt}, \forall L'' : \text{List}. [ & [\forall L' : \text{List}. \text{ map}(E ; L') = \text{map}(E) ; \text{map}(L')] \\ & \wedge \forall L' : \text{List}. \text{ map}(L'' ; L') = \text{map}(L'') ; \text{map}(L')] \\ \Rightarrow \forall L' : \text{List}. \text{ map}((E ; L'') ; L') & = \text{map}(E ; L'') ; \text{map}(L')] \end{aligned}$$

The first sentence can be easily proved by simplifying both sides of the equality with the equations in **MAP**. To prove the second sentence we can use the Theorem of Constants (see, e.g., [38]) to eliminate the outer quantification by turning the variables  $E$  and  $L''$  into constants  $\overline{E}$  and  $\overline{L''}$ . We can then prove the conclusion equation by simplifying both of its sides to a common expression using the equations in **MAP** and the induction hypothesis

$$(IH) \quad \text{map}(\overline{L''} ; L') = \text{map}(\overline{L''}) ; \text{map}(L')$$

as follows:

$$\begin{aligned} \text{map}((\overline{E} ; \overline{L''}) ; L') & \rightarrow_{\text{LIST}} \text{map}(\overline{E} ; (\overline{L''} ; L')) \rightarrow_{\text{MAP}} \\ f(\overline{E}) ; \text{map}(\overline{L''} ; L') & \rightarrow_{IH} f(\overline{E}) ; (\text{map}(\overline{L''}) ; \text{map}(L')) \end{aligned}$$

and

$$\text{map}(\overline{E} ; \overline{L''}) ; \text{map}(L') \rightarrow_{\text{MAP}} (f(\overline{E}) ; \text{map}(\overline{L''})) ; \text{map}(L') \rightarrow_{\text{LIST}} f(\overline{E}) ; (\text{map}(\overline{L''}) ; \text{map}(L'))$$

$\square$

Of course, the key remaining issue is to show that the inference system  $B[P] \vdash_{\text{ind}}$  is *sound*, that is, that for any  $\varphi \in \text{OSFOL}(\Sigma_B)$  we have:

$$B[P] \vdash_{\text{ind}} \varphi \quad \Rightarrow \quad B[P] \models_{\text{ind}} \varphi$$

---

<sup>5</sup> I am assuming that each connected component  $[s]$  of the poset of sorts  $(S, \leq)$  is finite.

**Theorem 11.** Let  $B[P]$  be a convergent parameterized specification that is sufficiently complete with respect to a subspecification  $C[P]$  of free constructors and such that: (i) the equations  $E_B$  are left-linear, and (ii) any constructor term  $t_0 \in T_{\Sigma_C}(X_P)$  is in  $E_B$ -canonical form. Then the inference system  $B[P] \vdash_{ind}$  is sound.

*Proof.* It is enough to show that  $B[P] \models_{ind} SInd(\forall x : s. \varphi)$  for each sentence of the form  $\forall x : s. \varphi \in OSFOL(\Sigma_B)$  such that  $s \in (S_B - S_P)$ . This is equivalent to proving that for each  $A \in \text{OSAlg}_P$  we have

$$F_J(A) \models SInd(\forall x : s. \varphi)$$

Since  $SInd(\forall x : s. \varphi)$  is an implication, we have to show that if the antecedent holds in  $F_J(A)$ , then the consequence holds. But by our assumptions on  $B[P]$  we have,  $F_J(A)_s = T_{(\Sigma_C(A) - \Sigma_P), s}$ , since all function symbols in  $\Sigma_P$  are eliminated by the rules  $\Delta(A)$ . Let us abbreviate  $T_{(\Sigma_C(A) - \Sigma_P), s}$  to  $Terms_s$ . We then obviously have,

$$Terms_s = \bigcup_{s' \prec s} Terms_{s'} \cup \bigcup_{c \in \Sigma_{C, s_1, \dots, s_n, s}} Terms(c)$$

where if  $c \in \Sigma_{C, s_1, \dots, s_n, s}$ , then  $Terms(c) = \{c(t_1, \dots, t_n) \mid t_i \in Terms_{s_i}\}$ . We now reason by contradiction. Suppose that the antecedent of  $SInd(\forall x : s. \varphi)$  holds for  $F_J(A)_s$  but the consequence fails. This means that there is a term  $t \in Terms_s$  of minimal depth such that  $\varphi(t)$  fails. Since the antecedent holds we cannot have  $t \in Terms_{s'}$  with  $s' \prec s$ . Therefore  $t$  must be of the form  $c(t_1, \dots, t_n)$  for some  $c \in \Sigma_{C, s_1, \dots, s_n, s}$ . And since  $t$  is of minimal depth, for each  $s_i \leq s$  we have  $\varphi(t_i)$ . But since the antecedent holds we then must have  $\varphi(c(t_1, \dots, t_n))$ , that is,  $\varphi(t)$  holds, a blatant contradiction.  $\square$

The above scheme of parameterized order-sorted structural induction could be formulated in other ways. For example, one could define an alternative scheme where instead of assuming  $\bigwedge_{s' \prec s} \forall x : s'. \varphi(x : s')$  we could assume  $\forall x : s''. \varphi(x : s'')$  for all  $s'' \in S_P$  such that: (i)  $s'' < s$ , and (ii)  $\exists s' \in S_P$  s.t.  $s'' < s' < s$ . And then we could consider not just all  $c \in \Sigma_{C, s_1, \dots, s_n, s}$ , but all  $c \in \Sigma_{C, s_1, \dots, s_n, s'}$  with  $s' \in (S_C - S_P)$ , and  $s' \leq s$ . The differences between both schemes are only pragmatic. The original scheme is more incremental and may be easier for human interaction, since it will tend to generate fewer goals, whereas the second scheme might be preferable when steps of structural induction plus attempts to prove the resulting subgoals are automated.

## 9 Stable Parameterized Induction

Parameterized inductive theorems of the form  $B[P] \models_{ind} \varphi$  are certainly quite useful, since they apply to all the free extensions  $F_J(A)$  for all  $A \in \text{OSAlg}_P$ . However, they are still quite limited.

The point is that usually what we need is not the theorem  $\varphi$ , but rather the theorem  $B[T] \models_{ind} H'(\varphi)$  for  $H : P \longrightarrow T$  a theory morphism instantiating the parameter theory  $P$ . The translation by  $H'$  is not essential, since for each  $A' \in \mathbf{OSAlg}_T$  we can use the “satisfaction condition” equivalence

$$F_{J'}(A') \models H'(\varphi) \Leftrightarrow U_{H'}F_{J'}(A') \models \varphi$$

to boil down the whole matter to a satisfaction problem for the  $\Sigma_B$ -algebras of the form  $U_{H'}F_{J'}(A')$ . So, what we now have to consider is a notion of *stable* parameterized inductive satisfaction.

**Definition 14.** (*Stable Parameterized Inductive Satisfaction and Theoremhood*). Let  $B[P]$  be a parameterized equational theory and let  $\mathcal{C}$  be a class of theory morphisms  $H : P \longrightarrow T$  that includes the identity theory morphism  $1_P : P \longrightarrow P$ . The  $\mathcal{C}$ -stable inductive satisfaction relation, denoted  $B[P] \models_{sind}^{\mathcal{C}} \varphi$ , is defined for each  $\varphi \in OSFOL(\Sigma_B)$  by the defining equivalence

$$B[P] \models_{sind}^{\mathcal{C}} \varphi \Leftrightarrow \mathbf{St}_{\mathcal{C}}\mathbf{FOSAlg}(B[P]) \models \varphi$$

where, by definition,  $\mathbf{St}_{\mathcal{C}}\mathbf{FOSAlg}(B[P]) = \{U_{H'}F_{J'}(A') \mid (H : P \longrightarrow T) \in \mathcal{C} \wedge A' \in \mathbf{OSAlg}_T\}$ . The set  $St_{\mathcal{C}}IndThm(B[P])$  of the  $\mathcal{C}$ -stable inductive theorems of  $B[P]$  is, by definition, the set of sentences

$$St_{\mathcal{C}}IndThm(B[P]) = \{\varphi \in OSFOL(\Sigma_B) \mid B[P] \models_{sind}^{\mathcal{C}} \varphi\}. \quad \square$$

Three useful special cases of  $\mathcal{C}$ -stability are: (i) the case where we have a free constructor specification  $C[P]$  and  $\mathcal{C}$  is the class of *all* theory morphisms  $H : P \longrightarrow T$ , which can be denoted  $C[P] \models_{sind} \varphi$ , (ii) the case when  $B[P]$  is stably convergent and sufficiently complete and  $\mathcal{C}$  is the class  $SD$  of those theory morphisms  $H : P \longrightarrow T$  such that  $H'(E_B)$  is sort-decreasing, which can be denoted  $B[P] \models_{sind}^{SD} \varphi$ , and (iii) the case when  $\mathcal{C}$  is the singleton set  $\{1_P : P \longrightarrow P\}$ , which is of course our old friend  $B[P] \models_{ind} \varphi$ .

Since  $\mathcal{C}$  always contains the identity morphism  $1_P : P \longrightarrow P$ , we always have a containment  $\mathbf{FOSAlg}(B[P]) \subseteq \mathbf{St}_{\mathcal{C}}\mathbf{FOSAlg}(B[P])$ . Therefore, we always have the implication

$$B[P] \models_{sind}^{\mathcal{C}} \varphi \Rightarrow B[P] \models_{ind} \varphi$$

and therefore the containment  $St_{\mathcal{C}}IndThm(B[P]) \subseteq IndThm(B[P])$ . The crucial question is whether such an implication is an equivalence, and therefore the above containment of theorems is an equality. For many-sorted algebras, when  $B[P]$  is persistent the answer to both questions is known to be *yes*, thanks to exactness, since we have isomorphisms  $F_JU_H(A') \cong U_{H'}F_{J'}(A')$  (see, e.g., 8.15 in [16]). But in order-sorted algebra we do not have exactness, and the implication is not in general an equivalence, as shown by the following example.

*Example 10.* Consider the following parameterized module **DISJOINT-UNION**, whose free extension semantics constructs the disjoint union of two sets in the parameter. Then consider its instantiation **COPY** by the view that merges the two different copies of **TRIV** in the parameter theory of **DISJOINT-UNION** into a single copy of **TRIV**.

```
fmod DISJOINT-UNION{X :: TRIV, Y :: TRIV} is
  sort U .  subsorts X$Elt Y$Elt < U .
endfm

fmod COPY{Z :: TRIV} is
  including DISJOINT-UNION{Z,Z} .
endfm
```

Obviously, **DISJOINT-UNION** is a parameterized free constructor specification. Therefore, by Theorem 7, **DISJOINT-UNION** is stably persistent. In particular, **COPY** is persistent. However, for  $H$  the theory morphism merging the  $X$  and  $Y$  copies of **TRIV** into the single copy  $Z$ , in general we do not have isomorphisms  $F_JU_H(A') \cong U_{H'}F_{J'}(A')$ , and the implication  $\text{DISJOINT-UNION} \models_{\text{sind}} \varphi \Rightarrow \text{DISJOINT-UNION} \models_{\text{ind}} \varphi$  is *not* an equivalence. For example, we have,

$$\text{DISJOINT-UNION} \models_{\text{ind}} \forall x : X\$Elt. \forall y : Y\$Elt. x \neq y$$

but

$$\text{COPY} \not\models_{\text{ind}} \forall x : Z\$Elt. \forall y : Z\$Elt. x \neq y.$$

That **COPY** does not satisfy the inductive parameterized inequality  $\forall x : Z\$Elt. \forall y : Z\$Elt. x \neq y$  becomes obvious as soon the parameter set is nonempty, since then we have an element  $a$  which must be equal to itself. In fact all models with a nonempty parameter set, not just those that are free extensions, fail to satisfy this inequality. It is also easy to see that **DISJOINT-UNION** does satisfy the inductive parameterized inequality  $\forall x : X\$Elt. \forall y : Y\$Elt. x \neq y$ . We can reason by contradiction. A model for the parameter is a pair of sets  $A = \{A_X, A_Y\}$ . Suppose that  $F_J(A)$  does not satisfy the inequality. This means that there are elements  $a \in A_X, b \in A_X$  such that  $\eta_A(a) = \eta_A(b)$ . But we also have a model of **DISJOINT-UNION** (not a free extension), which we can denote by 2, whose carrier set for each of the three sorts  $X\$Elt$ ,  $Y\$Elt$ , and  $U$  is the set  $\{0, 1\}$ . We can then define the  $P$ -homomorphism  $h : A \longrightarrow 2|_{\Sigma_P}$  mapping all elements of  $A_X$  to 0, and all elements of  $A_Y$  to 1. By the freeness of  $F_J(A)$  there is a unique **DISJOINT-UNION**-homomorphism  $\bar{h} : F_J(A) \longrightarrow 2$  such that  $\eta_A ; \bar{h} = h$ . But this means that  $\bar{h}(\eta_A(a)) = 0$ , and  $\bar{h}(\eta_A(b)) = 1$ , contradicting the equality  $\eta_A(a) = \eta_A(b)$ .  $\square$

The consolation prize question then becomes: is there a class of sentences  $OSFOL_0(\Sigma_B) \subseteq OSFOL(\Sigma_B)$  such that for each  $\varphi \in OSFOL_0(\Sigma_B)$  we indeed have an equivalence  $B[P] \models_{\text{sind}}^C \varphi \Leftrightarrow B[P] \models_{\text{ind}} \varphi$  for suitable  $B[P]$  and  $C$ ?

**Theorem 12.** (*Stability of Positive Inductive Theorems*). *Let  $B[P]$  be stably convergent and sufficiently complete with respect to a free constructor subspecification  $C[P]$ , and such that the equations  $E_B$  are left-linear and each term  $t_0 \in T_{\Sigma_C(F)}(X_F)$  is in  $!_P(E_B)$ -canonical form. Then for any universally quantified positive clause, that is, any sentence of the form*

$$\varphi = \forall x_1 : s_1, \dots, x_n : s_n. u_1 = v_1 \vee \dots \vee u_k = v_k$$

*we have the equivalence*

$$B[P] \models_{\text{sind}}^{SD} \varphi \Leftrightarrow B[P] \models_{\text{ind}} \varphi$$

*Proof.* We only need to prove the implication  $B[P] \models_{\text{ind}} \varphi \Rightarrow B[P] \models_{\text{sind}}^{SD} \varphi$ . Since it is well-known and easy to prove that if  $h : A \rightarrow B$  is a surjective homomorphism and  $A$  satisfies a universally quantified positive clause, then  $B$  must satisfy the same clause, it is enough to show that for each  $H : P \rightarrow T$  such that  $H'(E_B)$  is sort-decreasing and for each  $T$ -algebra  $A'$  there is a surjective  $\Sigma_B$ -homomorphism  $\delta_{A'} : F_J U_H(A') \rightarrow U_{H'} F_{J'}(A')$ . Indeed, if  $B[P] \models_{\text{ind}} \varphi$  we then have that for all  $H : P \rightarrow T$  such that  $H'(E_B)$  is sort-decreasing and for all  $T$ -algebras  $A'$ , since  $F_J U_H(A') \models \varphi$ , we must have  $U_{H'} F_{J'}(A') \models \varphi$ ; but this is just the definition of  $B[P] \models_{\text{sind}}^{SD} \varphi$ .

So, all we need to do is to define  $\delta_{A'} : F_J U_H(A') \rightarrow U_{H'} F_{J'}(A')$  and show that it is surjective. Note that by Theorem 10 we have a  $T$ -isomorphism  $\eta'_{A'} : A' \rightarrow U_J F_{J'}(A')$ , and therefore a  $P$ -isomorphism  $U_H(\eta'_{A'}) : U_H(A') \rightarrow U_H U_{J'} F_{J'}(A')$ . But since  $J; H' = H; J'$ , we have  $U_H U_{J'} F_{J'}(A') = U_J U_{H'} F_{J'}(A')$ , and therefore a  $P$ -isomorphism  $U_H(\eta'_{A'}) : U_H(A') \rightarrow U_J U_{H'} F_{J'}(A')$ , which by the freeness of  $F_J$  induces our desired  $\Sigma_B$ -homomorphism  $\delta_{A'} : F_J U_H(A') \rightarrow U_{H'} F_{J'}(A')$  such that  $\eta_{U_H(A)} ; U_J(\delta_{A'}) = U_H(\eta'_{A'})$ . We just need to show that  $\delta_{A'}$  is surjective. But since both  $\eta_{U_H(A)}$  and  $U_H(\eta'_{A'})$  are isomorphisms, the identity  $\eta_{U_H(A)} ; U_J(\delta_{A'}) = U_H(\eta'_{A'})$  forces  $\delta_{A',s}$  to be bijective for each  $s \in S_P$ , so we only need to worry about the surjectivity of  $\delta_{A',q}$  for each  $q \in (S_B - S_P)$ . That is, we have to show that for each  $q \in (S_B - S_P)$  and each  $t \in F_{J'}(A')_{H'(q)}$  there is a  $\tilde{t} \in F_J U_H(A')_q$  such that  $\delta_{A'}(\tilde{t}) = t$ . We can reason by contradiction and assume a term  $t \in F_{J'}(A')_{H'(q)}$  of minimal depth such that there is no term  $\tilde{t} \in F_J U_H(A')_q$  such that  $\delta_{A'}(\tilde{t}) = t$ . By Theorems 9 and 10, there must be a constructor  $c : s_1 \dots, s_n \rightarrow s$  in  $\Sigma_C$  with  $s \leq q$  such that our  $t \in F_{J'}(A')_{H'(q)}$  is of the form  $c(t_1, \dots, t_n)$ , with  $t_i \in F_{J'}(A')_{H'(s_i)}$ . But since  $t$  is of minimal depth failing the desired property, we have  $\tilde{t}_i \in F_J U_H(A')_{s_i}$  such that  $\delta_{A'}(\tilde{t}_i) = t_i$ ,  $1 \leq i \leq n$ . But since  $\delta_{A'}$  is a  $\Sigma_B$ -homomorphism and  $H'(c : s_1 \dots, s_n \rightarrow s) = c : H'(s_1) \dots, H'(s_n) \rightarrow H'(s)$ , we must have  $\delta_{A'}(c(\tilde{t}_1, \dots, \tilde{t}_n)) = c(t_1, \dots, t_n)$ , a blatant contradiction.  $\square$

*Example 11.* Since the inductive parameterized theorem

$$\text{MAP}\{\text{F} :: \text{FUNCTION}\} \vdash_{\text{ind}} \forall L, L' : \text{List}. \text{ map}(L; L') = \text{map}(L); \text{ map}(L')$$

proved in Example 9 is a positive clause, it is in fact a *SD-stable* theorem that is inductively true not only for  $\text{MAP}\{\text{F} :: \text{FUNCTION}\}$ , but for any  $\text{MAP}\{\text{T}\}$  such that  $H : \text{FUNCTION} \rightarrow T$  is such that  $H'(E_{\text{MAP}})$  is sort-decreasing.  $\square$

Since the above stability result only applies to positive clauses, a third question to ask is: what other parameterized inductive theorems are stable? In particular, what inductive theorems about *constructors* are stable?

**Proposition 3.** *Let  $C[P]$  be a parameterized specification of free constructors. Then the following parameterized inductive theorems are stable for all instantiations of  $P$ , where all variables are assumed universally quantified, and where we*

assume that  $c : s_1 \dots s_n \rightarrow q$  and  $c' : s'_1 \dots s'_n \rightarrow q'$  are two (not necessarily distinct) subsort-overloaded<sup>6</sup> constructors and  $c' : s''_1 \dots s''_n \rightarrow q''$  is another constructor in  $\Sigma_C - \Sigma_P$ , and where the sort  $s$  mentioned in the fourth sentence is a parameter sort  $s \in S_P$ , and the sort  $s_i$  in  $c : s_1 \dots s_n \rightarrow q$  is supposed to be in the same connected component as the sort  $q$ .

$$\begin{aligned} c(x_1:s_1, \dots, x_n:s_n) = c(y_1:s'_1, \dots, y_n:s'_n) &\Leftrightarrow x_1:s_1 = y_1:s'_1 \wedge \dots \wedge x_n:s_n = y_n:s'_n \\ c(x_1:s_1, \dots, x_n:s_n) \neq c(y_1:s'_1, \dots, y_n:s'_n) &\Leftrightarrow x_1:s_1 \neq y_1:s'_1 \vee \dots \vee x_n:s_n \neq y_n:s'_n \\ c(x_1:s_1, \dots, x_n:s_n) \neq c'(y_1:s''_1, \dots, y_n:s''_n) \\ y:s \neq c(x_1:s_1, \dots, x_n:s_n) \\ x_i:s_i \neq c(x_1:s_1, \dots, x_n:s_n) \end{aligned}$$

*Proof.* All these sentences follow from the stable persistence of  $C[P]$  (Theorem 7), and from the fact that for each theory morphism  $H \rightarrow T$ ,  $C[T]$  is itself a free constructor specification and for each  $T$ -algebra  $A'$ , the free extension  $F_{J'}(A')$  is precisely

$$F_{J'}(A) = T_{\Sigma_C(A')/\Delta(A')}|_{\Sigma_C} \cong \text{Can}_{\Sigma_C(A')/\Delta(A')}|_{\Sigma_C}$$

where  $\Sigma_C$  here means not the original signature  $\Sigma_{C[P]}$ , but the instantiated signature  $\Sigma_{C[T]}$ . Furthermore, since in  $\text{Can}_{\Sigma_C(A')/\Delta(A')}$  all function symbols in  $\Sigma_C - \Sigma_T$  have been eliminated, we have  $\text{Can}_{\Sigma_C(A')/\Delta(A')}|_{\Sigma_C - \Sigma_T} = T_{\Sigma_C(A') - \Sigma_T}$ , that is, a term algebra. Therefore, the first equivalence just follows from the fact that equality in a term algebra is exactly syntactic identity. The second equivalence is just a useful restatement of the first equivalence in negative form. The third inequality follows again from equality in a term algebra being syntactic identity. The fourth inequality follows from  $J' : T \hookrightarrow C[T]$  being a strongly guarded inclusion of signatures, so that a term of the form  $c(t_1, \dots, t_n)$  can never have a parameter sort  $H(s) \in S_T$ . The last inequality is an ‘‘occurs-check’’ type of inequality: no term can be equal to one of its strict subterms.  $\square$

**Corollary 4.** *Let  $B[P]$  be stably convergent and sufficiently complete with respect to a free constructor subspecification  $C[P]$ , and such that the equations  $E_B$  are left-linear and each term  $t_0 \in T_{\Sigma_{C[\mathbb{F}]}(X_{\mathbb{F}})}$  is in  $!_P(E_B)$ -canonical form. Then all the sentences in Proposition 3 are SD-stable parameterized inductive theorems of  $B[P]$ .*

The sentences in Proposition 3 can be very useful for *refutational purposes*. For example, we can easily refute the false conjecture  $\text{map}(E ; L) = \text{map}(L)$  using the equations for  $\text{map}$  and the last inequality in Proposition 3.

Is there anything else that can be said about the *SD*-persistence of negative universally quantified sentences of the form  $\varphi = \forall x_1:s_1, \dots, x_n:s_n. t \neq t'$ ? One last observation that can be made is the following proposition, which reduces reasoning about the *SD*-stable inequalities of  $B[P]$  to reasoning about such inequalities in a single and relatively simple free extension, namely, the free

<sup>6</sup> Recall that, by the assumptions about guarded signature inclusions, no ad-hoc overloading is permitted for function symbols in the body.

extension  $F_{J!}(1)$  below, where we can use standard inductive proof methods for unparameterized initial algebras.

**Proposition 4.** *For  $B[T]$  as in Theorem 12 and each sentence  $\varphi$  of the form  $\varphi = \forall x_1 : s_1, \dots, x_n : s_n. t \neq t'$  we have the equivalence:*

$$B[P] \models_{sind}^{SD} \varphi \Leftrightarrow F_{J!}(1) \models !'_P(\varphi)$$

where 1 is the only (up to isomorphism) model of the final theory  $\mathbb{F}$ , and  $J!$  is the theory inclusion  $J! : \mathbb{F} \hookrightarrow B[\mathbb{F}]$ .

*Proof.* The above equivalence follows from two simple observations: (i) If  $h : C \longrightarrow D$  is a homomorphism and  $\varphi = \forall x_1 : s_1, \dots, x_n : s_n. t \neq t'$ , then if  $D \models \varphi$  we necessarily have  $C \models \varphi$ , since otherwise we would have  $C \models \exists x_1 : s_1, \dots, x_n : s_n. t = t'$ , and a witness  $c$  for this existential formula would give us a witness  $c; h$  for it in  $D$ ; and (ii) for any  $SD$ -theory morphism  $H : P \longrightarrow T$ , and any  $T$ -algebra  $A'$  there is a  $\Sigma_B$ -homomorphism  $U_{H'}F_{J'}(A') \longrightarrow U'_{!P}(F_{J!}(1))$ . To see why this so, just notice that  $U_{!T}(1)$  is a final  $T$ -algebra. Therefore we have a unique  $T$ -homomorphism  $!_{A'} : A' \longrightarrow U_{!T}(1)$ , and therefore a composed  $T$ -homomorphism

$$A' \xrightarrow{!_{A'}} U_{!T}(1) \xrightarrow{U_{!T}(\eta_{!1})} U_{!T}U_{J!}(F_{J!}(1))$$

But since  $U_{!T}U_{J!}(F_{J!}(1)) = U_{J'}U'_{!T}(F_{J!}(1))$ , by the freeness of  $F_{J'}(A')$  this induces a  $\Sigma_{B[T]}$ -homomorphism  $F_{J'}(A') \longrightarrow U'_{!T}(F_{J!}(1))$ , which then gives us our desired  $\Sigma_B$ -homomorphism  $U_{H'}F_{J'}(A') \longrightarrow U'_{!P}(F_{J!}(1))$ .  $\square$

## 10 Related Work and Conclusions

There is a vast literature on parameterization of algebraic data types and, more generally, on parameterization in formal specification languages that I cannot hope to survey in this paper. A general bibliography on algebraic specifications up to 1991 can be found in [2]. I will restrict myself to giving a few sample references, to emphasize some developments, and to comment on work that I see as closely related to this paper.

I think that there is general agreement that abstract data type parameterization ideas came out of Burstall and Goguen's notion of using colimits in a category of theories to "put theories together" for modular purposes [7,8], from the ADJ group [55], and from a 1978 University of Dortmund report by H.-D. Ehrich, whose ideas became more widely available in [14]. Of course, they were all discovering the great importance for computer science of fundamental results on algebraic theories by Lawvere [35] and Bénabou [1], and advancing them further. For example, the full description of the Clear specification language [8] also hinted at the generalization of these ideas to other logics, which was later supported by the theory of institutions [22]. The free extension semantics of parameterized specifications was developed shortly afterwards, including the notion of persistence, e.g., [15,20,23]. The work of Padawitz [48,49] seems to be the

first to use rewriting techniques to ensure persistence, and to address the issue of parameterized induction. Later work on the use of rewriting techniques for parameterized inductive reasoning for many-sorted algebras includes, e.g., [34,4].

By comparison, the theory of parameterization for order-sorted algebras has developed more slowly. It was of course well-known from the early stages of order-sorted algebra that left adjoints to forgetful functors for theory morphisms existed, even for order-sorted Horn logic with equality [24]; and order-sorted parameterized data types were already used in OBJ2 [19]. But it was Poigné who first studied order-sorted parameterization in depth in his fundamental paper [50]. Poigné’s work made clear that order-sorted parameterization was considerably subtler than many-sorted parameterization, and found sufficient conditions within the OSA<sup>P</sup> formulation of order-sorted algebra (see Section 2) to ensure persistence of parameterized specifications by: (i) giving conditions for signature inclusions closely related to what I call guarded inclusions (see Definition 5), and (ii) restricting the instantiation of parameter theories to a special class of theory morphisms. However, Poigné’s work focused on what we might call the mathematical semantics of order-sorted parameterization and did not address either the use of rewriting techniques or inductive reasoning issues. After Poigné, Qian [51] and Haxthausen and Nickel [27] made additional contributions to order-sorted parameterization. More recently, order-sorted parameterization issues have also been investigated for the CASL language. The CASL order-sorted semantics is different: it is in a sense many-sorted, since, subsort inclusions are not interpreted as subset inclusions but as *injective coercion functions*. However, even with this more “permissive” semantics, the CASL institution still lacks exactness and therefore the amalgamation property. The problem is palliated in [52] by adopting an even more permissive semantics, in which subsort preorders become categories. Of course, the resulting notion of subsort becomes a bit tenuous, since now a “subsort” may have several different embeddings into a supersort.

Both rewriting techniques and inductive reasoning, as well as the mathematical semantics of parameterization, were later studied for special classes of parameterized membership equational theories by Bouhoula, Jouannaud and myself in [6]. Since membership equational logic contains order-sorted equational logic as a sublogic, this work shed some indirect light on order-sorted parameterization issues. However, although in theory persistence could be defined in the usual way as the isomorphism of the unit map, in practice such a definition would rule out almost all interesting membership equational logic examples. The issue is that as soon as new functions are added to the body of a parameterized specification, new *error terms* for that function typically appear at the kind level, making it impossible for the unit maps of free extensions to be isomorphisms. This required a weaker notion of persistence just at the level of *sorts*. Also, the issue of *stability* was not addressed in [6]. In summary, the two papers most closely related to this one are probably [50] and [6].

In conclusion, this paper has explored in depth the subtleties of order-sorted parameterization at three different levels: (i) its mathematical semantics; (ii)

its operational semantics by term rewriting; and (iii) the inductive reasoning principles that can soundly be used. It has also found reasonable conditions under which such specifications do indeed have the desired properties, as well as methods to check such conditions. As usual, much work remains ahead. The cases of parameterized specifications that are free *modulo* axioms such as associativity and/or commutativity and/or identity, and of conditional parameterized specifications should be investigated; more general sufficient conditions for persistence should be sought; and different parameterized inductive reasoning schemes beyond structural induction should be studied. Furthermore, parameterized inductive reasoning should be *mechanized*, for example as an extension of Maude's ITP tool [9,10,31]. This will integrate inductive reasoning for order-sorted parameterized specifications within the existing body of work, experience, and tools on inductive reasoning for order-sorted specifications, which, besides work on the Maude ITP [9,10,31], includes also work on proof scores in OBJ and CafeOBJ by Goguen and by Futatsugi and his collaborators [21,17], and the automata-driven approach of Bouhoula and Jouannaud [5].

**Acknowledgments.** I thank Camilo Rocha for his kind help in preparing the Latex diagrams of this paper and for his comments. I also thank Grigore Roşu, Ralf Sasse and an anonymous referee for their careful reading of the manuscript and their suggested improvements. This work has been partially supported by NSF Grants CNS 07-16638, IIS 07-20482, and CNS 08-34709.

## References

1. Bénabou, J.: Structures algébriques dans les catégories. *Cahiers de Topologie et Géometrie Différentielle* 10, 1–126 (1968)
2. Bidoit, M., Kreowski, H.-J., Lescanne, P., Orejas, F., Sannella, D. (eds.): Algebraic System Specification and Development. LNCS, vol. 501. Springer, Heidelberg (1991)
3. Bidoit, M., Mosses, P.D. (eds.): CASL User Manual. LNCS, vol. 2900. Springer, Heidelberg (2004)
4. Bouhoula, A.: Using induction and rewriting to verify and complete parameterized specifications. *Theor. Comput. Sci.* 170(1-2), 245–276 (1996)
5. Bouhoula, A., Jouannaud, J.-P.: Automata-driven automated induction. *Inf. Comput.* 169(1), 1–22 (2001)
6. Bouhoula, A., Jouannaud, J.-P., Meseguer, J.: Specification and proof in membership equational logic. *Theoretical Computer Science* 236, 35–132 (2000)
7. Burstall, R., Goguen, J.: Putting theories together to make specifications. In: Reddy, R. (ed.) *Proceedings, Fifth International Joint Conference on Artificial Intelligence*, pp. 1045–1058. Department of Computer Science, Carnegie-Mellon University (1977)
8. Burstall, R., Goguen, J.A.: The semantics of Clear, a specification language. In: Björner, D. (ed.) *Abstract Software Specifications*. LNCS, vol. 86, pp. 292–332. Springer, Heidelberg (1980)
9. Clavel, M.: *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications*. CSLI Publications (2000)

10. Clavel, M., Durán, F., Eker, S., Meseguer, J.: Building equational proving tools by reflection in rewriting logic. In: CAFE: An Industrial-Strength Algebraic Formal Method. Elsevier, Amsterdam (2000), <http://maude.cs.uiuc.edu>
11. Clavel, M., Durán, F., Eker, S., Meseguer, J., Lincoln, P., Martí-Oliet, N., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
12. Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree automata techniques and applications (2007), <http://www.grappa.univ-lille3.fr/tata> (Release, October 12th 2007)
13. Diaconescu, R., Goguen, J., Stefaneas, P.: Logical support for modularisation. In: Huet, G., Plotkin, G. (eds.) Logical Environments, pp. 83–130. Cambridge UP, Cambridge (1993)
14. Ehrich, H.-D.: On the theory of specification, implementation, and parametrization of abstract data types. *J. ACM* 29(1), 206–227 (1982)
15. Ehrig, H., Kreowski, H.-J., Thatcher, J.W., Wagner, E.G., Wright, J.B.: Parameter passing in algebraic specification languages. *Theor. Comput. Sci.* 28, 45–81 (1984)
16. Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification, vol. 1. Springer, Heidelberg (1985)
17. Futatsugi, K.: Verifying specifications with proof scores in CafeOBJ. In: ASE 2006, 21st IEEE/ACM International Conference on Automated Software Engineering, pp. 3–10 (2006)
18. Futatsugi, K., Diaconescu, R.: CafeOBJ Report. World Scientific, AMAST Series (1998)
19. Futatsugi, K., Goguen, J., Jouannaud, J.-P., Meseguer, J.: Principles of OBJ2. In: Reid, B. (ed.) Proceedings of 12th ACM Symposium on Principles of Programming Languages, pp. 52–66. ACM Press, New York (1985)
20. Ganzinger, H.: Parameterized specifications: Parameter passing and implementation with respect to observability. *ACM Trans. Program. Lang. Syst.* 5(3), 318–354 (1983)
21. Goguen, J.: OBJ as a theorem prover with application to hardware verification. In: Subramanyam, P., Birtwistle, G. (eds.) Current Trends in Hardware Verification and Automated Theorem Proving, pp. 218–267. Springer, Heidelberg (1989)
22. Goguen, J., Burstall, R.: Institutions: Abstract model theory for specification and programming. *Journal of the ACM* 39(1), 95–146 (1992)
23. Goguen, J., Meseguer, J.: Universal realization, persistent interconnection and implementation of abstract modules. In: Nielsen, M., Schmidt, E.M. (eds.) ICALP 1982. LNCS, vol. 140, pp. 265–281. Springer, Heidelberg (1982)
24. Goguen, J., Meseguer, J.: Models and equality for logical programming. In: Ehrig, H., Levi, G., Montanari, U. (eds.) TAPSOFT 1987 and CFLP 1987. LNCS, vol. 250, pp. 1–22. Springer, Heidelberg (1987)
25. Goguen, J., Meseguer, J.: Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science* 105, 217–273 (1992)
26. Goguen, J., Winkler, T., Meseguer, J., Futatsugi, K., Jouannaud, J.-P.: Introducing OBJ. In: Software Engineering with OBJ: Algebraic Specification in Action, pp. 3–167. Kluwer Academic Publishers, Dordrecht (2000)
27. Haxthausen, A.E., Nickl, F.: Pushouts of order-sorted algebraic specifications. In: Nivat, M., Wirsing, M. (eds.) AMAST 1996. LNCS, vol. 1101, pp. 132–147. Springer, Heidelberg (1996)

28. Hendrix, J., Meseguer, J.: On the completeness of context-sensitive order-sorted specifications. In: Baader, F. (ed.) RTA 2007. LNCS, vol. 4533, pp. 229–245. Springer, Heidelberg (2007)
29. Hendrix, J., Meseguer, J.: Order-sorted equational unification revisited. *Electr. Notes Theor. Comput. Sci.* (2008); To appear in Proc. of RULE 2008
30. Hendrix, J., Meseguer, J., Ohsaki, H.: A sufficient completeness checker for linear order-sorted specifications modulo axioms. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 151–155. Springer, Heidelberg (2006)
31. Hendrix, J.D.: Decision Procedures for Equationally Based Reasoning. PhD thesis, University of Illinois at Urbana-Champaign, UIUCDCS-R-2008-2999 (2008), <http://www.cs.uiuc.edu/research/phdtheses.php>
32. Hullot, J.-M.: Canonical forms and unification. In: Bibel, W. (ed.) CADE 1980. LNCS, vol. 87, pp. 318–334. Springer, Heidelberg (1980)
33. Kirchner, C., Kirchner, H., Meseguer, J.: Operational semantics of OBJ3. In: Lepistö, T., Salomaa, A. (eds.) ICALP 1988. LNCS, vol. 317, pp. 287–301. Springer, Heidelberg (1988)
34. Kirchner, H.: Proofs in parameterized specifications. In: Book, R.V. (ed.) RTA 1991. LNCS, vol. 488, pp. 174–187. Springer, Heidelberg (1991)
35. Lawvere, F.W.: Functorial semantics of algebraic theories. *Proceedings National Academy of Sciences* 50, 869–873 (1963); Summary of Ph.D. Thesis, Columbia University
36. MacLane, S.: Categories for the Working Mathematician. Springer, Heidelberg (1971)
37. Meseguer, J.: General logics. In: Logic Colloquium 1987, pp. 275–329. North-Holland, Amsterdam (1989)
38. Meseguer, J.: Membership algebra as a logical framework for equational specification. In: Parisi-Presicce, F. (ed.) WADT 1997. LNCS, vol. 1376, pp. 18–61. Springer, Heidelberg (1998)
39. Meseguer, J., Goguen, J., Smolka, G.: Order-sorted unification. *J. Symbolic Computation* 8, 383–413 (1989)
40. Mosses, P.: Action Semantics. Cambridge University Press, Cambridge (1992)
41. Mosses, P.D.: A basic abstract semantic algebra. In: Plotkin, G., MacQueen, D.B., Kahn, G. (eds.) Semantics of Data Types 1984. LNCS, vol. 173, pp. 87–107. Springer, Heidelberg (1984)
42. Mosses, P.D.: Unified algebras and action semantics. In: Cori, R., Monien, B. (eds.) STACS 1989. LNCS, vol. 349, pp. 17–35. Springer, Heidelberg (1989)
43. Mosses, P.D.: Unified algebras and institutions. In: Proc. Fourth Annual IEEE Symp. on Logic in Computer Science, Asilomar, California, June 1989, pp. 304–312 (1989)
44. Mosses, P.D.: Unified algebras and modules. In: POPL, pp. 329–343 (1989)
45. Mosses, P.D.: The use of sorts in algebraic specifications. In: Bidoit, M., Choppy, C. (eds.) COMPASS/ADT 1991. LNCS, vol. 655, pp. 66–92. Springer, Heidelberg (1991)
46. Mosses, P.D. (ed.): CASL Reference Manual. LNCS, vol. 2960. Springer, Heidelberg (2004)
47. Mosses, P.D.: Modular structural operational semantics. *J. Log. Algebr. Program.* 60–61, 195–228 (2004)
48. Padawitz, P.: Parameter-preserving data type specifications. *J. Comput. Syst. Sci.* 34(2/3), 179–209 (1987)
49. Padawitz, P.: The equational theory of parameterized specifications. *Inf. Comput.* 76(2/3), 121–137 (1988)

50. Poigné, A.: Parametrization for order-sorted algebraic specification. *J. Comput. Syst. Sci.* 40(2), 229–268 (1990)
51. Qian, Z.: Another look at parameterization for order-sorted algebraic specifications. *J. Comput. Syst. Sci.* 49(3), 620–666 (1994)
52. Schröder, L., Mossakowski, T., Tarlecki, A., Klin, B., Hoffman, P.: Amalgamation in the semantics of CASL. *Theor. Comput. Sci.* 331(1), 215–247 (2005)
53. Smolka, G., Nutt, W., Goguen, J., Meseguer, J.: Order-sorted equational computation. In: Nivat, M., Aït-Kaci, H. (eds.) *Resolution of Equations in Algebraic Structures*, vol. 2, pp. 297–367. Academic Press, London (1989)
54. Strachey, C., Milne, R.: *A Theory of Programming Language Semantics*. Chapman and Hall, Boca Raton (1976)
55. Thatcher, J.W., Wagner, E.G., Wright, J.B.: Data type specification: Parameterization and the power of specification techniques. In: STOC 1978 Tenth Annual ACM Symposium on Theory of Computing, pp. 119–132. ACM Press, New York (1978)

# An Implementation of Object-Oriented Action Semantics in Maude<sup>\*</sup>

André Murbach Maidl<sup>1</sup>, Cláudio Carvilhe<sup>2</sup>, and Martin A. Musicante<sup>3,4,\*\*</sup>

<sup>1</sup> Programa de Pós-Graduação em Informática,  
Universidade Federal do Paraná, Curitiba, Brazil  
[andremm@gmail.com](mailto:andremm@gmail.com)

<sup>2</sup> Departamento de Informática,  
Pontifícia Universidade Católica do Paraná, Curitiba, Brazil  
[carvilhe@ppgia.pucpr.br](mailto:carvilhe@ppgia.pucpr.br)

<sup>3</sup> Programa de Pós-Graduação em Sistemas e Computação,  
Universidade Federal do Rio Grande do Norte, Natal, Brazil

<sup>4</sup> LIFO - Laboratoire d’Informatique Fondamentale d’Orléans,  
Université d’Orléans, France  
[mam@dimap.ufrn.br](mailto:mam@dimap.ufrn.br)

**Abstract.** We present *Maude Object-Oriented Action Tool*, an executable environment that incorporates elements from Object-Oriented and Constructive Action Semantics. Our tool is implemented as a conservative extension of Full Maude and Maude MSOS Tool. The syntax used by our tool is fairly similar to the one used by the original Action Semantics formalism. Furthermore, we present an Object-Oriented Action Semantics library of classes, capable of supporting constructive (object-oriented) action semantics.

## 1 Introduction

In more than 20 years of development, the Action Semantics framework has evolved in many different directions. The original design by Mosses and Watt [19,20,27] has been used as a basis for the proposition of a number of variants of action semantics. Most of these variants deal with the introduction of modularity to the framework.

Altough Action Semantics inherently presents good reusability, the standard Action Notation lacks of syntactic support for the definition of libraries and reusable components [13]. In order to overcome this problem, a modular approach for Action Semantics has been proposed in [10], where Action Semantics *modules* allow the use of isolated specification elements as well as module composition.

Based on the modular approach, introduced by Doh and Mosses in [10], Object-Oriented Action Semantics (OOAS) has been proposed in [4] as a method

---

<sup>\*</sup> This work is an extended, revised version of [15].

<sup>\*\*</sup> Partly supported by CAPES, Brazil (BEX 3164/08-0) and CNPq, Brasil (Grant 473071/2007-0.).

to organize Action Semantics specifications by introducing the notion of *class*. OOAS was defined using SOS [25] transitions in [4].

As reported by [26], the lack of tools for describing programming languages semantics (using the object-oriented formalism) is a problem. Our goal is to supply additional resources to OOAS, providing a tool where it is possible to write OOAS specifications and execute them, and giving a view of how OOAS specifications may be achieved in a constructive way.

In this regard, we propose MOOAT (Maude Object-Oriented Action Tool): the first tool for describing programming languages semantics using OOAS, completely described and available at [16]. MOOAT development was inspired by MAT (Maude Action Tool) [2] and its implementation using a Modular SOS [22] interpreter, which was developed as a Rewriting Logic [17] semantic framework in Maude version 1 [7]. In order to achieve a better modularity and ease of use of our tool, we propose some changes in OOAS syntax and semantics. One important difference between the tool and the original OOAS formalism is that MOOAT semantics is defined by using Modular SOS [22] instead of plain SOS.

We have used the Maude system [7] to implement the Classes Notation of OOAS and its Action Notation has been implemented using the Modular SOS environment provided by MMT (Maude MSOS Tool) [5,6], mostly in accordance with the Modular SOS for Action Notation proposed in [21].

Besides the implementation, we propose a library of Object-Oriented Action semantics classes, called LFLv2 (read LFL version 2; LFL stands for Language Features Library<sup>1</sup>). This library we incorporates some of the ideas presented in [14,23], in order to achieve the separation of the syntax of the programming language from the specification of its semantics.

In this work, we present two case studies, to be seen as proof of concepts: **LFLv2** and **COOAS**. The former one is the implementation of LFLv2, populated with some basic classes. The latter one presents a *constructive* approach for Object-Oriented Action Semantics. Basically, we combine Constructive Action Semantics [14,23] with Object-Oriented Action Semantics.

Both case studies were developed to increase the modularity aspects observed in the object-oriented formalism and also to obtain a new syntax-independent style for describing programming languages.

This work is organized as follows: the next section briefly introduces Maude MSOS Tool. Section 3 gives an overview of Object-Oriented Action Semantics and Language Features Library. Constructive Action Semantics is summarized in section 4. LFLv2, the new version of the library of classes is proposed in section 5. MOOAT notation and implementation are explained in section 6. In section 7 we discuss how Language Features Library version 2 as well as Constructive Object-Oriented Action Semantics descriptions are obtained as case studies of MOOAT. The conclusions of this work as well as future work are exposed in section 8.

---

<sup>1</sup> A first version of the library was proposed in [1], but it proved to have some problematic aspects in its design.

## 2 Maude MSOS Tool

Structural Operational Semantics (SOS) [25] is a formal framework extensively used to specify programming languages and other frameworks, such as Action Semantics [20]. However, modularity in SOS specifications was left open by Plotkin in [25]. The modularity problem in SOS specifications has been treated by Mosses' Modular SOS (MSOS) [22].

MSOS uses a general transition system, where components like environments and storage are implemented by labeled transitions instead of being part of configurations, in order to improve modularity. In Plotkin's SOS, configurations may be syntactic trees, computed values or auxiliary entities. Transitions, in SOS are defined by expressions of the form " $C_1 \rightarrow C_2$ ", where  $C_1$  and  $C_2$  are configurations.

In MSOS, configurations always will be just syntactic trees or computed values. Any necessary auxiliary entities will be included as part of a label. Transitions, in MSOS are defined by expressions of the form " $T_1 \xrightarrow{\alpha} T_2$ ", where  $T_1$  and  $T_2$  are syntactic trees or evaluated values and  $\alpha$  is a *label*.

The label  $\alpha$  in a transition represents all the information associated to this transition, including the current environment and storage state, before and after the computation described by the transition.

Such labels are seen as category morphisms that have composition operations. The labels category is usually the product category and a notation is provided to access and change specific components independently.

In labels, the pattern `{...}` is used to represent a completely arbitrary label. A pattern such as `{env=Env,...}` allows getting information from a specific component of a label without mentioning other components; and a pattern such as `{sto=Store,sto'=Store',...}` allows changing a specific component of a label; in this case a new memory position (`Store'`) might be added to the current store (`Store`). When it is not necessary to refer to label components, they can be simply omitted.

Maude MSOS Tool (MMT) [5,6] is an executable environment to Modular SOS specifications. MMT is a formal tool implemented as a conservative extension of Full Maude [11] that compiles MSOS specifications into Rewriting Logic [5,17].

The syntax adopted by MMT is based on the Modular SOS Definition Formalism (MSDF) created by Mosses to be used in MSOS specifications. Both languages are fairly similar, in this way, those that use MSOS probably would easily understand a MSOS specification in MMT. Nonetheless, small differences exist between them due to peculiarities in the Maude parser [6,7] - for instance, the conflict among operators from Maude built-in modules and those present in the language that is being specified.

MMT is very useful for the formal specification of programming languages and formal frameworks since it is possible to define the abstract syntax using BNF and give a semantics for that syntax by defining a set of labeled transitions containing the necessary semantic components. Also, it is possible to organize the specification into modules, guaranteeing the construction of a modular specification that presents good reusability.

### 3 Object-Oriented Action Semantics

In Object Oriented Action Semantics (OOAS) [4], an *object* encapsulates some Action Semantics features. Class constructors and several other object-based operators are offered in extension to the standard Action Notation, providing an object-oriented way of composing specifications.

In this section we present OOAS by examples, using a simple language of commands:

```
Command ::= Identifier “:=” Expression | Command “;” Command |
           “if” Expression “then” Command ‘else’ Command “end-if” |
           “while” Expression “do” Command
```

The previous BNF definition establishes that commands can be conditionals, assignments, iterations or sequences. The first step using OOAS is to define the *Abstract Class* (or base class). In a simple way, the main non-terminal symbol can be *elected* as such a class. Therefore, the following class is obtained:

```
Class Command
  syntax:
    Cmd
  semantics:
    execute _ : Cmd → Action
End Class
```

**Command** is the abstract class. The **syntax** section introduces the syntactic tree **Cmd**. The **semantics** section introduces the semantic function **execute**, establishing a mapping from commands to actions. In OOAS, semantic functions are seen as *methods*. Now, every particular **Command** behavior should be defined. The tactic is to define any specific command as a *specialized class*, as follows:

```
Class While
  extending Command
  using E:Expression, C:Command
  syntax:
    Cmd ::= “while” E “do” C
  semantics:
    execute [] “while” E “do” C [] =
    unfolding
      evaluate E then
      | execute C and then unfold
        else complete
End Class
```

Notice that **While** is a subclass of **command**. Reusability can be achieved employing object-orientation concepts like inheritance and class hierarchy. The **extending** directive states the meaning of a particular command (Iteration). The **using**

directive allows us to reuse existing classes (as in *E*:Expression and *C*:Command). In the *semantics* section, *While* is specified, as a plain Action Semantics *action*.

OOAS has shown to be a practical approach to modularity. OOAS Notation is simple, inspired by the notions from object-oriented programming and similar to the original Action Notation. Instantiation and extension permit the construction of libraries of programming languages concepts, improving reusability. In fact, OOAS Notation is a mixture of the original Action Notation with the Class Notation created by the object-based approach. OOAS semantics has been specified by SOS rules and it has been reported in [4].

In OOAS, the semantics description of a given programming language is a hierarchy of classes. In this regard, all defined classes belong to a pre-defined root class called *State*. Such class is the base-class for all OOAS classes and it is responsible to implement the attributes and operations that are used in those classes specifications.

All the classes defined in the object-oriented formalism are sub-classes of *State*, turning visible both attributes and operations to its sub-classes. The *State* attributes are concerned with the information processed by actions, such as: *transients*, *bindings* and *storage*. Some operations are available to handle *State* attributes. Such operations are all basic actions and action combinators.

The behavior of OOAS descriptions is similar to the original Action Semantics and can be classified into facets, as defined in [20]: **basic** (deals with pure control flow); **functional** (deals with actions that process transient data); **declarative** (deals with actions that produce or receive bindings); **imperative** (deals with actions that manipulate the storage); **reflective** (deals with abstractions); **hybrid** (deals with actions from more than one facet). No operation was defined to deal with the **communicative** facet.

A complete OOAS description as well as its SOS specification can be found in [4]. In addition to it, an OOAS library of classes has been proposed in [1], intensively using this method and it is presented in what follows.

### 3.1 Language Features Library

LFL [1] is concerned as a library of generic specification classes constructed using OOAS. The idea is to concentrate a set of common specification features into a repository (or library), so it can be available to new languages' specifications.

A tree structure was adopted to represent the class organization in LFL. LFL was branched off into three main classes: *Syntax*, *Semantics* and *Entity*. The node *Semantics* is forked into another three nodes: *Declaration*, *Command* and *Expression*. Pieces of code that manipulate bindings are treated in *Declaration*; *Command* represents semantic definitions that are concerned with data flow and the storage; while the classes for manipulation of values belong to *Expression*.

Each one of the above-defined nodes was split into two: *Paradigm* and *Shared*. The first one is responsible to represent classes that contain features from a specific programming language paradigm, such as: Object-Oriented, Functional, Logical and Imperative. The node *Shared* contains classes which represent features that are common to more than one programming paradigm.

The following example illustrates the definition of a LFL class. This class defines the semantics of a while command:

Class While

```
<< Command implementing < execute _ : Command → Action >
   Expression implementing < evaluate _ : Expression → Action > >>
locating LFL.Semantics.Command.Shared
using E:Expression, C:Command
semantics:
  execute-while(E, C) =
    unfolding
    | evaluate [ E ] then infalibly select
    | given true then execute [ C ] then unfold
    | or
    | given false then skip
```

End Class

The While class provides the semantics of a while-loop command in a syntax-independent style. The classes **Command** and **Expression** are passed as (higher order) parameters. Notice that the generic arguments must comply with the restrictions defined by the implementing directive. They must provide some methods with a specific type as sub-parameters.

The directive **locating** identifies the place where the class is located in the LFL structure.

The programming language specifications which use the LFL are similar to the plain Object-Oriented Action Semantics descriptions. Let us now exemplify the use of the generic While class to define the semantics of a while command<sup>2</sup>:

Class MyCommand

syntax:

Com

semantics:

myexecute \_ : Com → Action

End Class

Class MyWhile

extending MyCommand

using E:MyExpression, C:MyCommand,

objLoop:LFL.Semantics.Command.Shared.While <<

MyCommand<myexecute>, MyExpression<myevaluate> >>

syntax:

Com ::= "while" E "do" C

semantics:

myexecute [ "while" E "do" C ] =  
objLoop.execute-while(E, C)

End Class

<sup>2</sup> The user's side of the specification should contain new names for the classes, like MyCommand and MyWhile specified here.

The classes `MyCommand` and `MyWhile`, as defined above, specify the behavior of a while-loop command using the library of classes by instantiating LFL's generic class `While`. The arguments for this LFL class are the `MyCommand` and `MyExpression` classes. Notice that their respective sub-parameters, `myexecute` and `myevaluate` methods, are also provided.

LFL brings an interesting concept to Object-Oriented Action Semantics: the definition of semantic descriptions which are independent from the syntax of the programming language. This definition uses the inclusion of classes provided by a library of classes. A main issue with the way in which LFL is defined is that its parameter-passing mechanism is obscure and let us back to the readability problems found in other semantic frameworks.

In the following section we turn our attention to Constructive Action Semantics, a constructive approach concerned with code reusability and focused on the formal specification of programming languages separating syntax from semantics.

## 4 Constructive Action Semantics

It is common to find semantically similar constructors in different programming languages, even if these constructors have very different syntax. In this regard, it is interesting to reuse parts of the programming language specifications that represent the same behavior. The constructive approach introduced in [23], helps with the code reuse by supporting independent definitions and using named modules to describe individual languages features.

Constructive Action Semantics [14,23] is based on the idea that each language feature is defined by a separate and independent basic abstract construct. The semantics of a complete language is achieved by translating its constructs into the basic abstract constructs. That is, the main idea of this approach consists in mapping concrete language constructs to combinations of basic abstract constructs.

Concrete constructs are related to the way programming languages implement their features while abstract constructs are concerned in representing these features using a language-independent prefixed notation. For instance, a while-loop concrete syntax could be written in the following way: `while (Exp) do Cmd;` and its respective abstract syntax could be translated to: `cond-loop(Exp, Cmd).`

Notice that the concrete construct of the while-loop is composed by an expression (the loop condition) and a command (the loop body). The concrete syntax shown above inhibits the parsing ambiguity among the constructs while the corresponding abstract syntax is used just to differentiate one construct from other constructs.

A huge variety of constructs is present in the developed programming languages and this is the reason why they might be considered as basic or derived. Basic constructs represent common features that have the same interpretations and are found in many programming languages. The constructs that have a specific behavior regarding the programming language are classified as derived

constructs, they are included in few languages and specified by combining basic constructs.

The modular approach of Action Semantics has been used with the constructive approach in order to obtain a high degree of modularity in Constructive Action Semantics, as discussed in [14]. In this related work, the Action Semantics Definition Formalism (ASDF) has been specially designed to write Action Semantics descriptions of single language constructs using Action Semantics modules.

Now, we will see how the examples shown in section 3 may be written in the formalism presented in this section. The ASDF notation has been used to specify the necessary modules to implement an individual basic abstract construct for a while-loop command.

```
Module Cmd
requires C:Cmd
semantics execute : Cmd → Action
```

The above module defines a variable  $C$  from a sort `Cmd` which will be used to specify basic abstract constructs for commands. A semantic function called `execute` is also defined to describe the implemented basic abstract construct using Action Semantics.

```
Module Cmd/While
syntax Cmd ::= cond-loop(Exp, Cmd)
requires Val ::= Boolean
semantics execute cond-loop( $E$ ,  $C$ ) =
  unfolding
    evaluate  $E$  then
      execute  $C$  and then unfold else complete
```

In the module `Cmd/While` we have a derived module from `Cmd` module. Notice that the individual basic abstract construct `cond-loop(Exp, Cmd)` was used to specify the semantics of a while-loop command with a boolean condition independently of its syntax. In this way, such a module could be used in any project that needs a while-loop. The specification of a whole language in Constructive Action Semantics is achieved by combining modules that implement the necessary basic individual constructs into a single module.

We have introduced the constructive approach proposed in [23], focused on its usage with Action Semantics. However, it also can be used with Modular SOS. The concrete constructs of Core ML have been translated to basic constructs as a Constructive Action Semantics case study in [14] and the formalism has been successfully validated to describe programming languages semantics syntax independently. Motivated by these results, in section 7.2 we will present how constructive semantics can be applied to Object-Oriented Action Semantics using the tool described in section 6. The next section introduces how the library of classes issues might be solved.

## 5 Language Features Library Version 2

LFL is a good approach for specifying programming languages semantics syntax independently since it has support for generic classes definitions [1]. Nonetheless, the obscure LFL syntax with parameters passing took us back to the readability problems found in semantic frameworks. An alternative way to inhibit parameters passing, while instantiating library classes in a formal specification which uses LFL, is the use of abstractions of actions to specify the semantics of some common concepts of programming languages.

Furthermore, we propose the LFL usage just for semantic concepts. It implies in the exclusion of the LFL nodes: *Syntax*, *Semantics* and *Entity*. Thereby, we can bind the nodes *Declaration*, *Command* and *Expression* directly to the main LFL class since we will use LFL just to represent semantic concepts. The nodes *Shared* and *Paradigm* are maintained, as well as their respective subdivisions. The former is concerned with constructs that commonly appear in programming languages and the latter represents some specialized constructs that appear in specific programming paradigms.

As in the first version of the library, the set of classes is very reduced and can be extended to support new classes. According to our proposal, the *While* class example shown in section 3.1 would be written in the following way:

```

Class While
  locating LFL.Command.Shared.While
  using Y1:Yielder, Y2:Yielder
  semantics:
    execute-while(Y1,Y2) =
      unfolding
      |enact Y1 then
      ||enact Y2 and then unfold else complete
End Class

```

Now, LFL classes are more compact since they just define their own location in the LFL hierarchy, the methods that describe the semantics of some concepts of programming languages and the objects used in the specified methods.

Instead of using methods that have to be passed as parameters, each method implemented by the new library of classes receives a number of abstractions that are performed according to the language behavior that is being represented. The *enact* action receives *yielders* that result in abstractions (and encapsulate actions). These actions are performed independently from the methods described in the programming language specification. Let us now see how to use the new LFL:

```

Class MyCommand
  syntax:
    Com
  semantics:
    myexecute _ : Com → Action
End Class

```

**Class MyWhile**

extending MyCommand

using  $E$ :MyExpression,  $C$ :MyCommand

$objLoop$ :LFL.Semantics.Command.Shared.While

syntax:

$Com ::= \text{"while"}~E~\text{"do"}~C$

semantics:

$\text{myexecute}~[\text{[}]~Com ::= \text{"while"}~E~\text{"do"}~C~\text{[}] =$

$objLoop.\text{execute-while}(\text{closure abstraction of (myevaluate } E),$   
 $\text{closure abstraction of (myexecute } C))$

End Class

Now, MyExpression and MyCommand classes do not need to be passed as arguments when the instance of  $objLoop$  is created. It is possible due to the fact that execute-while method is able to process the *abstractions* created by calling closure abstraction of with the methods defined in the specified language. In this way, language methods are used only by its own specification, while the *abstractions* are performed by the library without any dependency of user's methods.

The notation  $\text{encapsulate } X$  has been created in order to provide the *abstractions* used by the new classes of LFL and it will be used in the rest of this paper. It is an abbreviation of closure abstraction of ( $X$ ). Notice that this clause ought to be used when specifying languages with static bindings. If the language being described has dynamic bindings, the directives *closure* should be taken from the specification.

The point of LFLv2 is on creating *abstractions* using the methods defined on the language specification. It means that user's methods are not passed as parameters to the library. Actually, they are translated to Object-Oriented Action Notation. Since the *actions* are in shape of *abstractions*, it is possible to either interpret or execute them using the *action enact* provided by the reflective facet.

## 6 Maude Object-Oriented Action Tool (MOOAT)

In this section we present MOOAT<sup>3</sup>, a tool that provides an executable environment to the formal specification of programming languages using Object-Oriented Action Semantics. It has been developed using MMT [5] to implement the Modular SOS of Action Notation, introduced by Mosses in [21], and also using Maude [7] to implement the ideas proposed by [4].

### 6.1 Notation

A formal specification in Object-Oriented Action Semantics is a finite set of classes. Such classes create the necessary hierarchy to represent the formal specification of a language or library. The relationship among these classes is defined according to the instantiated objects, as well as the position where classes are found in the specified hierarchy.

---

<sup>3</sup> The current implementation of MOOAT can be downloaded from  
<http://www.ppgia.pucpr.br/~carvilhe/mooat/>.

The described tool is a conservative extension of Full Maude [11] and MMT [6]. In this way, programming languages can be described in Object-Oriented Action Semantics by MOOAT, while Full Maude and MMT still might be used in connection with Rewriting Logic [17] and Modular SOS [22], respectively.

Nevertheless, the environment has the same limitations of Maude and MMT, as explained in [5,7,16]. We shall cite: *pre-regularitycheck*, that is the requirement in which a term must have one least sort [5]; and *ad-hoc overloading*, which requires that if the sorts in the arities of two operators with the same syntactic form are pairwise in the same connected components, then the sorts in the coarities must likewise be in the same connected component [7].

Due to these limitations MOOAT syntax has some differences from OOAS syntax, yet both are very similar. MOOAT notation is given using BNF as follows:

- (1) *ClassModule* ::= “class” *ClassName* “is”  
    ⟨ *ClassExtends*⟩\* ⟨ *ClassDefinition*⟩\* “endclass”
- (2) *ClassExtends* ::= “extends” *ClassName* ⟨ “,” *ClassName*⟩\* “.”
- (3) *ClassDefinition* ::= ⟨ *SyntacticPart*⟩\* ⟨ *SemanticPart*⟩\*

The class structure is defined in the rules (1) to (3). A class begins with the directive `class` and ends with `endclass`. The body class is composed basically by the declaration of base classes and by the class definition. Notice that more than one class can be specified as a base class.

The notation used by our tool is a variation of the OOAS notation introduced in [4]. Amongst the differences between these notations we can point out the changing of the directive extending to `extends` and the exclusion of the directive `using`. The former was done because the directive extending already exists in Maude and if we had redefined it we would have the *pre-regularity* problem discussed in [5]. The latter was done owing to the fact that now object declarations are automatically included in the methods definition, based on the idea of MSDF metavariables, described in [5,6].

The syntax for the body of each class module is defined by rules (4) to (11), as follows:

- (4) *SyntacticPart* ::= *SyntacticSort* “.” | *SyntacticSort* “::=” *syntax-tree* “.”
- (5) *SemanticPart* ::= ⟨ *SemanticFunctions*⟩\* ⟨ *SemanticEquations*⟩\*
- (6) *SemanticFunctions* ::= ⟨ *SemanticFunction*⟩<sup>+</sup>
- (7) *SemanticFunction* ::=  
        “absmethod” *FunctionName* *Tokens* “->” “Action” “.” |  
        “absmethod” *FunctionName* *Tokens* “->” *Data* “.”
- (8) *Tokens* ::= *TokenName* ⟨ “,” *Tokens*⟩\*
- (9) *SemanticEquations* ::= ⟨ *SemanticEquation*⟩<sup>+</sup>
- (10) *SemanticEquations* ::=  
          “method” *FunctionName* *syntax-tree-with-objects* “=” *Action* “.”  
          “method” *FunctionName* *syntax-tree-with-objects* “=” *Data* “.”
- (11) *ObjectDeclaration* ::= *Identifier* “.” *SyntacticSort*

Like in the original Object-Oriented Action Semantics, the body class is composed by two parts: syntax and semantics. The syntactic part follows the BNF

introduced by MMT. The semantic part may be composed by semantic functions, which will be called as abstract methods, and semantic equations, which will be called as simply methods.

An abstract method works as a signature of a method that will be specified in a class or in a sub-class and may result in an action or in a data sort. A method implements the semantics of a programming language concept using actions and action combinators. Methods must be used to give the semantics of the abstract syntax defined in the class. When such abstract syntax is used in the method declaration, syntactic sorts must be changed by object declarations.

The automatic object creation concept is introduced, since the syntactic sorts are changed by identifiers that represent these syntactic sorts in the methods implementations. These objects will be used to perform encapsulated actions and to give the desired semantic meaning.

Rules (12) to (16), given below, define the Action Notation to be used in the methods body:

- (12) *Action* ::= *Action* “or” *Action* | “fail” | “commit” |  
*Action* “and” *Action* | “complete” | “indivisibly” *Action* |  
*Action* “and then” *Action* | *Action* “trap” *Action* |  
“escape” | “unfolding” *Action* | “unfold” | “diverge” |  
“give” *Yielder* | “regive” | “choose” *Yielder* |  
“check” *Yielder* | *Action* “then” *Action* | “escape with” *Yielder* |  
“bind” *Yielder* “to” *Yielder* | “rebind” | “unbind” *Yielder* |  
“produce” *Yielder* | “furthermore” *Action* |  
*Action* “moreover” *Action* | *Action* “hence” *Action* |  
*Action* “before” *Action* | “store” *Yielder* “in” *Yielder* |  
“unstore” *Yielder* | “reserve” *Yielder* |  
“unreserve” *Yielder* | “enact” *Yielder* |  
“indirectly bind” *Yielder* “to” *Yielder* | “indirectly produce” *Yielder* |  
“redirect” *Yielder* “to” *Yielder* | “undirect” *Yielder* |  
“recursively bind” *Yielder* “to” *Yielder* |  
*Action* “else” *Action* | “allocate” *Yielder*
- (13) *Yielder* ::= *Data* | “a” *Yielder* | “the” *Yielder* | “nothing” |  
“the” *DataSort* “yielded by” *Yielder* | “it” | “them” |  
“given” *DataSort* | “given” *DataSort* # *Int* |  
“current bindings” | “the” *DataSort* “bound to” *Yielder* |  
*Yielder* “receiving” *Yielder* |  
“current storage” | “the” *DataSort* “stored in” *Yielder* |  
“application” *Yielder* “to” *Yielder* | “closure” *Yielder* |  
“encapsulate” *Action* | “indirect closure” *Yielder*
- (14) *Data* ::= *Datum* | *DataSort*
- (15) *Datum* ::= “none” | “unknown” | “uninitialized” | *Abstraction* |  
“<” *Int* “>” | “<” *Boolean* “>” | “<” *Token* “>” | “<” *Cell* “>” |  
“<” *Transients* “>” | “<” *Bindings* “>” | “<” *Storage* “>” |
- (16) *DataSort* ::= “integer” | “truth-value” | “token” | “cell” |  
“abstraction” | “value”

A coercion function (`< >`) must be used with the implemented data due to the *ad-hoc overloading* problem cited in [7]. Notice that the available actions and action combinators are represented in rule (12) while rule (13) represents the available yielders that can be used to produce data during an action performance. The last rules represent the available sorts and sorts of data. To exemplify the use of MOOAT we will see how the classes shown in section 3 are written:

```
(class Command is
  Cmd .
  absmethod execute Cmd -> Action .
endclass)
```

In the above example we have the abstract class `Cmd` which just defines the syntactic sort `Cmd` and the abstract method `execute` that maps a command to an action.

```
(class While is
  extends Command .
  Cmd ::= while Exp do Cmd .
  method execute (while E:Exp do C:Cmd) = unfolding
    ((evaluate E:Exp) then
      ((execute C:Cmd and then unfold) else complete)) .
endclass)
```

The `While` class is the sub-class of `Cmd` and a while-loop is implemented in it. The syntactic sort `Cmd` is overloaded with the command syntax and the semantics is given in the overloaded method `execute`. The objects `E:Exp` and `C:Cmd` were used to give the while semantics, that is, the command in the object `C:Cmd` is executed until the expression in the object `E:Exp` is true.

Notice that, previously an object was declared by an identifier and its respective class in the directive `using`. However, now it is directly declared by an identifier and its respective syntactic sort in the method definition. In this way we have introduced the automatic object creation concept.

In the next section we will see how the Modular SOS for Action Notation was implemented using MMT and how the Classes Notation was specified in Maude.

## 6.2 Implementation

MOOAT has been built as a conservative extension of Full Maude and MMT. Action Notation is given by MSDF modules which implement the available actions through Modular SOS transitions. Some Maude operations and equations have been implemented to support the Classes Notation proposed by the object-oriented style of Object-Oriented Action Semantics.

This implementation was possible since the modularity in Modular SOS specifications is in the labels used by transitions. Therefore, using MMT it was possible to implement the Action Notation modules defined in [21] and the Classes Notation which supports specifications in the OOAS style.

First of all, we will show how Action Notation transitions were implemented as well as Data Notation. After that we will elucidate how the Maude built-in module (LOOP-MODE [5,7,11]) which deals with input and output was changed to accept OOAS classes and translate them into Maude system modules, the same technique employed in object-oriented modules of Full Maude [11] and MSDF modules of MMT [5]. Then we will mention how *State* class, detailed in section 3, is treated.

Data Notation has been specified by reusing datatypes that are provided by both Maude and MMT. For instance, the sorts **Int** and **Boolean** defined by MMT were used to represent integers and truth-values respectively. However, some coercion functions were needed to avoid the *ad-hoc overloading* problem in these datatypes specification. It can be observed in the rule (15) in section 6.1.

The abstract syntax of Action Notation has been specified in MMT using datatype declarations due to the extended-BNF syntax provided by it. Mixfix operations might be defined in order to specify the abstract syntax of the language or formalism that is being defined, as it is shown in the following module:

```
(msos BasicSyntax is
  see BasicData .

  Action ::= Action or Action | fail | commit |
            Action and Action | complete |
            indivisibly Action |
            Action and'then Action |
            Action trap Action | escape |
            unfolding Action | unfold | diverge .

  Yielder ::= the DataSort yielded'by Yielder |
             nothing | Data .
sosm)
```

Basic facet abstract syntax is being implemented by the MSDF module **BasicSyntax**. Notice that the sorts **Action** and **Yielder** were extended to define the *actions* and *yielders* available in basic facet, and also that data components are regarded as already evaluated.

Action performance and yielder evaluation may compute values. These values can be defined algebraically in MMT as sorts, operations and predicates, as well as it is shown in the following module:

```
(msos FunctionalOutcomes is
  see BasicOutcomes .
  see FunctionalData .

  Completed .

  Terminated ::= Completed .
```

```

Completed ::= completed .
Completed ::= gave (Data) .

gave (none) : Action --> completed .
sosm)

```

Notice that the MSDF modules `BasicOutcomes` and `FunctionalData` were included in `FunctionalOutcomes`. In such module the `Terminated` sort is redefined to accept the values introduced by the sort `Completed`, which are: `completed`, to determine that an action is completed, and `gave (Data)`, to specify a transient data production. When no data is given then the action is simply completed.

```

(msos BasicConfigurations is
 see BasicSyntax .
 see BasicOutcomes .

Action ::= Terminated | Action @ Action .
sosm)

```

In the MSDF module `BasicConfigurations` we have introduced that configurations for non-distributed action performance are always the same. The sort `Terminated` is related to the results produced by actions and it may depend on the facet. An auxiliary construct (`Action1 @ Action2`) was defined to be used just by the basic facet and with the `unfolding` construct.

Each facet of Action Notation usually requires the implementation of labels to treat with the facet's components. For instance, functional facet treats with transient data, in this way its label have to carry the current transient data produced by the current actions.

```

(msos FunctionalLabels is
 see BasicLabels .
 see FunctionalData .

Label = {data : Transients, data' : Transients, ...} .
sosm)

```

The label for the functional facet were implemented in the MSDF module `FuncionalLabels` as a read-write label. Its indexes `data` and `data'` carry the current transient data since they are being represented by the sort `Transients`, which is in fact a map from a positive number to a simple datum.

Now we will see some of the implemented transitions needed to understand the MOOAT development process. Basically, we have three main kinds of transitions.

Those transitions that neither change or use the labels' components. Such as shown in the following example which defines an interleaved performance of “`Action1` and `Action2`”.

```
Action1 -{...}-> Action'1
```

---

```
Action1 and Action2 : Action -{...}-> Action'1 and Action2 .
```

```
Action2 -{...}-> Action'2
```

---

```
Action1 and Action2 : Action -{...}-> Action1 and Action'2 .
```

Those transitions that just use the labels' components. In the following example we have a rule that takes the transient data carried by the component **data** and turn them available for the next actions.

```
regive : Action -{data = Transients,
           data' = Transients,-}-> gave (< Transients >) .
```

Those transitions that change and use the labels' components. Now we have an example were the label component has been changed, beyond it has been used. The transient data **Data1** and **Data2** are processed and the current transient data, represented by **Transients**, are changed to have them as the new transient data, represent by **Transients''**.

```
Transients' := (1 |-> Data1) / Transients,
Transients'' := (2 |-> Data2) / Transients'
```

---

```
gave (Data1) and gave (Data2) : Action
  -{data = Transients, data' = Transients'',-}->
    gave (concatenation(Data1, Data2)) .
```

In MOOAT, the *State* class has been represented partially by a system module called **STATE** and partially by the Classes Notation that will be detailed from here.

The *State* class of MOOAT has five attributes instead of only three as we mentioned in section 3. Such attributes are used in the specification of programming languages and have been represented by labels' indexes implemented by the Action Notation described above.

Since we are describing an implementation using MMT, those attributes must be initialized as the initial configuration needed to compute an action. Such configuration has been represented by the command **compute** which has been defined in **STATE** module receiving an action as parameter. This command must be used with Full Maude's command **rewrite** in order to init the MRS [6,18] configuration implemented by MMT and used by MOOAT to create the executable environment that computes an action.

*State* attributes are classified and initialized according to the following table:

Attribute	Data Flow	Initial Value
commitment	a boolean value for <i>commit</i>	false
unfolding	an action	fail
data	the transient data	empty map ( <i>Int</i> → <i>Datum</i> )
bindings	the bindings	empty map ( <i>Token</i> → <i>Data</i> )
storage	the storage	empty map ( <i>Cell</i> → <i>Data</i> )

Notice that in MOOAT, the operations that act over the main class are the transitions implemented by each one of the implemented facets. In this way, the actions provided by the Object-Oriented Action Semantics of MOOAT are quite similar to the actions provided by the Modular SOS for Action Notation described in [21], as it was proposed as a further work in [4].

The technique employed in the development of the Classes Notation is similar to the used on the construction of object-oriented modules in Full Maude [11] and on the definition of MSDF modules in MMT [6]. That is, a specific syntax is created and when it is used its constructions are translated to a code that Maude is able to interpret.

MOOAT classes are composed basically by three parts: extends part, syntactic part and semantics part. The use of these parts might be optional or sequential. In other words, they may be not used, used just once or more than once. On the other hand, a class must have at least one of these three parts. The definition of an empty class is not allowed.

Classes in MOOAT were implemented as alternative MSDF modules where just the three parts mentioned before are accepted in a class definition. The definition of a sub-class is simply translated to lines that use the Maude's directive `including`. In the syntactic part we have reused the definition of syntax trees in the BNF style implemented by MMT; these trees and syntactic sorts are translated, respectively, to operations, sorts and subsorts of Maude.

The methods in the semantic part are translated to a set of equations supported by Maude system modules. While the objects defined automatically are treated as meta-variables from Maude in those equations set. As well as in Object-Oriented Action Semantics, in MOOAT every class is a direct sub-class of *State*. In this way, when we create a new class the main class is automatically included and makes available the defined Action Notation. For the fact that a class is converted to a system module, the *State* class was implemented directly as a system module. For this reason it is possible to add it in every converted class.

A methods environment is created by the inclusion of the converted MOOAT classes into the Maude's module database. The MOOAT root class provides the necessary operations to change its attributes and the environment for the methods definition that is still being read just by the operations previously defined. However, when a new class is added its respective methods are also added.

### 6.3 Related Work

There is a considerable number of tools that has been built for defining the syntax and semantics of programming languages, as well as to support the development of such language definitions. However, in this section, we will focus on the discussion of tools based on Action Semantics.

The Actress system [3] was developed to interpret Action Notation and compile it into C code. Most of the Action Notation is covered by Actress, except the communicative facet.

The ABACO system [24] is an Action Semantics tool developed focusing on teaching students how to design programming languages. It is a very interesting and complete tool since it enables to build and test Action Semantics descriptions through a graphical user interface. ABACO also has an algebraic specification compiler, specification editors, action libraries and action editors. The system can be used to process Action Semantics descriptions or even generate interpreters and compilers.

The Action Environment [26] was designed to offer more flexibility while dealing with modular specifications, based on the Constructive Action Semantics approach. It has been implemented on top of the ASF+SDF Meta-Environment [12] and has been introduced ASDF (Action Semantics Definition Formalism) for writing Action Semantics descriptions of single language constructs. The tool's key point is in supporting reuse of descriptions of individual constructs.

Another tool based on the ASF+SDF Meta-Environment is the ASD tools [9] which provided a textual syntax-directed. It supported editing, parsing, checking and interpreting Action Semantics descriptions.

MAT [2,8] was implemented as a prototype for Action Semantics specifications. Its Action Notation was written using a Modular SOS interpreter which translated the code into Rewriting Logic in Maude. Later, the earlier MSOS interpreter motivated the creation of MMT [5,6], a new powerful tool for writing MSOS descriptions that helped us on the creation of our execution engine.

Even though MOOAT is one in a line of tools for execution based on Action Semantics, before there was no environment for specifying programming languages using Object Oriented Action Semantics and it has been our main motivation for building such tool.

In this section we have presented some aspects from the tool's implementation. Such implementation consists of the adaptation of the Action Notation present in the original Object-Oriented Action Semantics SOS rules to MSOS rules introduced by Mosses and supported by MMT. Moreover, the Classes Notation were implemented using the powerful system provided by Maude. The implementation aspects were summarized in this paper to introduce the use of MOOAT and also to the comprehension of its implementation. For more details about it we shall indicate [16] as the main source.

## 7 MOOAT Case Studies

In section 3 we have presented Object-Oriented Action Semantics, an approach for language definition using Action Semantics with object-oriented concepts and based on the modularity in Action Semantics by splitting descriptions into classes.

We also have introduced a tool that supports the definition of programming languages or libraries in Object-Oriented Action Semantics. Hence, we will show two MOOAT case studies: the first one is the implementation of LFL version 2 while the second addresses the creation of Constructive Object-Oriented Action Semantics.

In both case studies we will see how a programming language would be defined using them. For this reason, we will present the specification of a toy language called  $\mu$ -Pascal. This language is fairly similar to Pascal language;  $\mu$ -Pascal is an imperative programming language containing basic commands and expressions. Its respective syntax was defined using BNF in [16].

## 7.1 LFLv2

In this section we present how LFL version 2, proposed on section 5, can be implemented as a MOOAT case study. Using MOOAT, it has been possible to test library's modularity aspects as well as its new structure. By the following examples we try to demonstrate how modularity and readable aspects has been enhanced on LFL and also that it might be used to specify programming languages in a syntax independent way.

The examples related to commands, from section 5, are presented in the tool's syntax to introduce the proposed case study.

```
(class LFL.Command.Shared.While is
  extends LFL.Command.Shared .
  absmethod execute-while (Yielder, Yielder) -> Action .
  method execute-while (Y1:Yielder, Y2:Yielder) = unfolding
    ((enact Y1:Yielder) then
     ((enact Y2:Yielder and then unfold) else (complete))) .
  endclass)
```

In spite of using formal framework locating directive, its content is used as class name. This has been done in order to preserve the hierarchy proposed by LFL.

The class `LFL.Command.Shared.While` is the implementation of the class `While` presented in section 5. As well as the formal class, this one is also tied to the node `LFL.Command.Shared` by the use of `extends`, completing the hierarchy definition. The method `execute-while` is implemented in the same way it has been proposed in section 5. We shall remind that using MOOAT is necessary to defined abstract methods before defining the method itself. As explained on section 6, object declarations are given automatically when defining a method.

```
(class MyCommand is
  Com .
  absmethod myexecute Com -> Action .
  endclass)

(class MyWhile is
  extends MyCommand .
  extends LFL.Command.Shared.While .
  Com ::= while Exp do Com .
  method myexecute (while E:Exp do C:Com) =
    execute-while (encapsulate (myevaluate E:Exp),
                  encapsulate (myexecute C:Com)) .
  endclass)
```

The class `MyWhile` is quite similar to that one specified in section 5. The class `LFL.Command.Shared.While` is extended in order to access the method `execute-while` which receives *abstractions* as parameters and describes a while-loop semantics syntax independently. Such *abstractions* are created using `encapsulate` and are used by `myexecute` method to call library's loop. In this way, the *action* might be performed outside the language definition and independently of the methods defined by the user.

Now we will see how  $\mu$ -Pascal would be defined using LFLv2.

```
(class Micro-Pascal is
  Prog .
  Prog ::= begin Dec ; Com end .
  absmethod run Prog -> Action .
  method run (begin D:Dec ; C:Com end) =
    myelaborate D:Dec hence myexecute C:Com .
  endclass)
```

First of all, the syntactic sort `Prog` is defined and the syntax which defines the structure of a  $\mu$ -Pascal program is assigned to it. After that, the abstract method `run` is defined to map a program to an *action*. Then, such *action* is performed by elaborating declarations before executing commands. Once `myelaborate` or `myexecute` are called, they interpret the syntax carried by `D:Dec` and `C:Com` and then call the necessary LFLv2 methods to process them.

In these regards, the class `Micro-Pascal` defines the formal semantics of  $\mu$ -Pascal using LFLv2 and the definition of the programming language is given in a syntax independent way since the language defined by the user just specify its syntax while the semantics is defined by the use of library's methods.

The classes related to commands were presented in this section to clarify how new LFL can be used with MOOAT. LFLv2, its full set of classes and complete implementation examples can be found in [16]. The set of classes that can be found is related to implemented classes to each one of the LFLv2 nodes: *Declaration*, *Command* and *Expression*.

We would like to highlight the fact that the library allows defining a language independently of its syntax, since the semantical definition is related to the methods provided by the library. Such methods might be used to define several languages no mattering the syntax. It is possible due to using abstractions of actions which are performed by the library to specify the semantical behavior for a certain feature of the language that is being defined.

## 7.2 COOAS: A Constructive Approach to OOAS

In section 4 we have presented Constructive Action Semantics, a constructive approach related to Action Semantics and based on the idea of a collection of basic abstract constructs that may be used in different programming languages projects. In this section we show that is possible to combine the formalisms described in sections 3 and 4, in order to obtain Constructive Object-Oriented Action Semantics as a MOOAT case study.

To be more specific, using MOOAT we will demonstrate that the modularity aspects observed in the object-oriented approach might be improved by adding the constructive ideas into it. Furthermore, an approach with good modularity, easy readable and that helps on the programming languages specifications syntax independently can be achieved.

Again we will use examples related to commands in order to introduce the proposed ideas.

```
(class Cmd is
  Cmd .
  absmethod execute Cmd -> Action .
endclass)
```

The class `Cmd` works as an abstract class since it just introduces the syntactic sort `Cmd`, which will be used in the definition of the commands constructs that will be implemented, and the abstract method `execute` to work as a semantic function. Notice that the presented class is exactly the same if compared to its respective class in section 6.

```
(class Cmd/While is
  extends Cmd .
  Cmd ::= cmd-while (Exp, Cmd) .
  method execute (cmd-while(E:Exp, C:Cmd)) = unfolding
    (evaluate E:Exp then
      ((execute C:Cmd and then unfold) else complete)) .
endclass)
```

In the `Cmd/While` class we have a specialized class of `Cmd` that implements the construct of a while-loop, `cmd-while (Exp, Cmd)`. Notice that the main difference between Object-Oriented Action Semantics and Constructive Object-Oriented Action semantics is in overloading the syntactic sort with a basic abstract construct instead of using the language concrete construct. This basic abstract construct is also used when the method `execute` is overloaded in order to give its semantics.

We have created a set of classes for Constructive Object-Oriented Action Semantics in [16]. Those classes were specified in the same style as `Cmd` and `Cmd/While` to deal with expressions, commands, declarations, values and programs.

For expressions constructs we have `Exp` class and its respective sub-classes that deal with arithmetical and logical values and expressions. As in the examples mentioned in this section, the class `Cmd` and its sub-classes are related to commands constructs. The class `Dec` introduces declarations constructs which will be used in its specialized classes of declarations. A class to specify a program construct was also implemented and it was called `Prog`.

Below we see how  $\mu$ -Pascal would be defined using COOAS classes.

```
(class Micro-Pascal is
  extends Exp/Val, Exp/Val-Id .
  extends Exp/Sum, Exp/Sub, Exp/Prod .
  extends Exp/True, Exp/False, Exp/LessThan, Exp/Equality .
  extends Cmd/Assignment, Cmd/Repeat, Cmd/While .
  extends Cmd/Sequence, Cmd/Cond .
  extends Dec/Variable, Dec/DecSeq .
  extends Prog .
endclass)
```

The concrete constructs of  $\mu$ -Pascal are represented by each class specified in the `extends`' lines. Such classes implement the basic abstract constructs needed by the specified programming language. The reason why we do not have reference to the abstract classes is that they are already referenced by their own sub-classes.

In these regards, the class `Micro-Pascal` defines the formal semantics of  $\mu$ -Pascal, using Constructive Object-Oriented Action Semantics, just extending the specialized classes that were designed to give the semantics of specific features. The definition of the programming language was achieved independently of its syntax for the fact that those features were implemented by basic abstract constructs as it has been proposed in [23].

Like in Constructive Action Semantics, in Constructive Object-Oriented Action Semantics we have to translate the concrete syntax of the language to its respective combination of basic abstract constructs. This was done in the example above and any language that has similar features to  $\mu$ -Pascal could be specified in this way.

### 7.3 MOOAT Case Studies Remarks

In this section we presented two case studies that were used to test MOOAT. Both LFLv2 and COOAS implemented  $\mu$ -Pascal language. The purpose of implementing such simple language was to experiment the approaches using MOOAT. Using the tool it is possible to play with specifications in the object-oriented formalism within the objective to test programming languages or new approaches based on code reuse.

Before starting to use MOOAT we suggest to first define either the language will be specified using purely OOAS or a modular approach like LFLv2 and COOAS. The initial set of classes for LFLv2 and COOAS is pretty similar. While LFLv2 contains ten classes for Expressions, COOAS defines eleven classes<sup>4</sup>. Both approaches have five classes for Commands and four classes for Declarations.

Altough the number of classes is not that huge it is important to remember that it can be easily extended. Either LFLv2 or COOAS set of classes can be edited in order to insert new classes that implement other concepts of programming languages.

---

<sup>4</sup> COOAS implements one more class for Expressions which is used to evaluate a single value. Since it is transparent in LFLv2 this is the reason for the difference.

Regarding performance, COOAS has been shown lighter than LFLv2. The result of a factorial of five is obtained with COOAS in three minutes while with LFLv2 it takes eight minutes to run. The difference is explained by the way the language is defined.

As we have seen in this section, a language defined using COOAS only extends classes that are already defined, not concerning about the syntax of the language which is being described. The syntax will be defined apart from the semantics specification. On the other hand, LFLv2 requires a more complex hierarchy of classes. When specifying a programming language using LFLv2, each specialized class must define the syntax of the concept that is being implemented beyond its semantics.

## 8 Conclusions and Future Work

Action Semantics and Object Classes are relevant concepts which can be used on the formal specification of languages and systems. OOAS is a framework where the combination of these two concepts is used. Previously to our work, no executable environment for writing and executing OOAS specifications had been provided. MOOAT has been developed considering this demand and is based on an earlier (from other authors) Action Semantics tool called MAT, which was developed using an earlier MSOS interpreter developed in Maude.

In this paper we have presented MOOAT, the first implementation of Object-Oriented Action Semantics. Our first idea consisted of developing an isolated OOAS tool from scratch. However, it seemed impractical knowing the existence of MAT. The better choice, in our view, was creating an extension of MMT. The idea was to aggregate the object-oriented apparatus to a brand new tool based on MAT and developed using MMT. In MOOAT, the user can create OOAS specifications and perform any necessary tests inside the standard Maude environment. This is interesting, considering that the user can create Maude, MMT, and MOOAT specifications using the same tool.

This implementation has contributed to the formal specification of OOAS since a Modular SOS definition has been provided to its Action Notation using MMT and the Maude system has been used to define its Classes Notation. It means that the previous specification of OOAS using SOS has been rewritten using MSOS and implemented using the MSOS language provided by MMT. Also, that a language to specify OOAS classes has been built in Maude.

A practical result of our work is that now it is easier to update and insert new features to OOAS due to the MSOS specification of MOOAT. Such specification has modular aspects that helps to extend what is written by adding new elements. Even being written using MMT, MOOAT is the update of OOAS since its formal specification has been translated from SOS to MSOS. Also, the fact of using Maude and MMT to develop MOOAT provided the first executable environment for OOAS specifications. Furthermore, MOOAT notation covers a rich set of actions and action combinators, which includes the complete OOAS notation.

Besides the implementation, a new and improved version of LFL has been proposed to solve the issues with parameters passing found in its first version. Such problems have been solved by using abstractions of actions instead of passing classes and methods as parameters to LFL classes. In this way, LFL became more concise and simplified due to the use of reflective facet.

We also have combined Constructive Action Semantics (CAS) with Object-Oriented Action Semantics (OOAS) in order to achieve Constructive Object-Oriented Action Semantics (COOAS). This constructive semantics represents a novel view to the Object-Oriented Action Semantics system.

The introduction of constructs in OOAS has contributed to improve the modularity aspects observed in the object-oriented approach. In our view, the constructive approach can be largely employed with code reuse when adopted in existent formalisms, as it happened with MSOS, AS and now OOAS. This is the first time that both formalisms are combined. Notice that COOAS has been shown using MOOAT, although it could be used as a specification approach independently of the tool usage, like LFLv2 does.

Both LFLv2 and COOAS contribute to improve the modularity aspects observed in the object-oriented approach. Moreover, both ideas are capable to describe syntax-independent specifications of programming languages. In [16], MOOAT development is described in more details as well as case studies are deeply presented. As future work we would implement the **communicative** facet of Action Notation. Since Constructive Object-Oriented Action Semantics is a rigorous analysis subject, we should present it in more details as well as a careful comparison between LFLv2 and COOAS in the future.

## Acknowledgments

We would like to thank MMT authors, Christiano Braga and Fabricio Chalub, for the interest in this work and for the helpful support while we were using their tool to develop MOOAT. We are most grateful to the anonymous referees for their careful review.

## References

1. Araújo, M., Musicante, M.A.: Lfl: A library of generic classes for object-oriented action semantics. In: XXIV International Conference of the Chilean Computer Science Society (SCCC 2004), Arica, Chile, November 11-12, 2004, pp. 39–47. IEEE Computer Society Press, Los Alamitos (2004)
2. Braga, C., Haeusler, E.H., Meseguer, J., Mosses, P.D.: Maude action tool: Using reflection to map action semantics to rewriting logic. In: Rus, T. (ed.) AMAST 2000. LNCS, vol. 1816, pp. 407–421. Springer, Heidelberg (2000)
3. Brown, D.F., Moura, H., Watt, D.A.: Actress: an action semantics directed compiler generator. In: Pfahler, P., Kastens, U. (eds.) CC 1992. LNCS, vol. 641, pp. 95–109. Springer, Heidelberg (1992)
4. Carvilhe, C., Musicante, M.A.: Object-oriented action semantics specifications. Journal of Universal Computer Science 9(8), 910–934 (2003)

5. Chalub, F., Braga, C.: Maude msos tool. Technical report, Universidade Federal Fluminense (2005), <http://maude-msos-tool.sourceforge.net/mmt-manual.pdf>
6. Chalub, F., Braga, C.: Maude msos tool. In: Denker, G., Talcott, C. (eds.) Proceedings of 6th International Workshop on Rewriting Logic and its Applications, WRLA, Vienna, Austria. Elsevier, Amsterdam (2006)
7. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: Maude manual, version 2.3 (2007),  
<http://maude.cs.uiuc.edu/maude2-manual/html/>
8. Braga, C.d.O., Haeusler, E.H., Meseguer, J., Mosses, P.D.: Maude action tool: Using reflection to map action semantics to rewriting logic. In: Rus, T. (ed.) AMAST 2000. LNCS, vol. 1816, pp. 407–421. Springer, Heidelberg (2000)
9. Van Deursen, A., Mosses, P.D.: Asd: The action semantic description tools (1996)
10. Doh, K.-G., Mosses, P.D.: Composing programming languages by combining action-semantics modules. In: van den Brand, M., Parigot, D. (eds.) LDTA 2001. ENTCS, vol. 44.2. Elsevier, Amsterdam (2001)
11. Durán, F., Meseguer, J.: The Maude specification of Full Maude. Technical report, SRI International (1999)
12. Klint, P. (ed.): The asf+sdf meta-environment. Technical report, CWI, Centrum voor Wiskunde en Informatica, Amsterdam (1992),  
<ftp://ftp.cwi.nl/pub/gipe/reports/SysManual.ps.Z>
13. Labra Gayo, J.E.: Reusable semantic specifications of programming languages. In: SBLP 2002 - VI Brazilian Symposium on Programming Languages (2002)
14. Iversen, J.: Formalisms and tools supporting Constructive Action Semantics. PhD thesis, BRICS International PhD School (May 2005)
15. Maidl, A.M., Carvilhe, C., Musicante, M.A.: Maude object-oriented action tool. Electr. Notes Theor. Comput. Sci. 205, 105–121 (2008)
16. Maidl, A.M.: A maude implementation of object-oriented action semantics. Master's thesis, Universidade Federal do Paraná (2007) (in portuguese)
17. Martí-Oliet, N., Meseguer, J.: Rewriting logic as a logical and semantic framework. Technical report, SRI International (1993)
18. Meseguer, J., Braga, C.d.O.: Modular rewriting semantics of programming languages. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) AMAST 2004. LNCS, vol. 3116, pp. 364–378. Springer, Heidelberg (2004)
19. Mosses, P.D.: Action semantics. In: ADT (1986)
20. Mosses, P.D.: Action Semantics. Cambridge Tracts in Theoretical Computer Science, vol. 26. Cambridge University Press, Cambridge (1992)
21. Mosses, P.D.: A modular SOS for Action Notation. BRICS RS 99-56, Dept. of Computer Science, Univ. of Aarhus (1999)
22. Mosses, P.D.: Modular structural operational semantics. J. Logic and Algebraic Programming 60-61, 195–228 (2004); Special issue on SOS
23. Mosses, P.D.: A constructive approach to language definition. Journal of Universal Computer Science 11(7), 1117–1134 (2005)
24. Moura, H., Menezes, L.C., Monteiro, M., Sampaio, P., Cansanção, W.: The abaco system: An action tool for programming language designers. In: AS 2002, pp. 1–8. Dept. of Computer Science (2002) BRICS NS-02-8
25. Plotkin, G.D.: A structural approach to operational semantics. J. Logic and Algebraic Programming 60-61, 17–139 (2004); Special issue on SOS
26. van der Brand, M., Iversen, J., Mosses, P.D.: An action environment. Sci. Comput. Program. 61(3), 245–264 (2006)
27. Watt, D.A., Thomas, M.: Programming language syntax and semantics. Prentice Hall International (UK) Ltd, Hertfordshire (1991)

# A Constructive Semantics for Basic Aspect Constructs

Christiano Braga<sup>\*</sup>

Universidade Federal Fluminense, Brazil  
cbraga@ic.uff.br

**Abstract.** Peter Mosses has contributed to computer science in many different ways. In particular, to programming language semantics. I had the pleasure, and the honor, to work with Peter Mosses as a PhD student and to collaborate with him afterwards. His work has greatly influenced my research interests. In this paper, I focus on his constructive approach to the semantics of programming languages. Constructive Semantics is an approach to the specification of programming language semantics that focuses on *reuse*. As new programming paradigms are developed, the library of reusable parts of semantic components provided by the constructive approach may be extended with new constructs to support the new paradigm. I propose constructs to support basic aspect-oriented programming concepts together with their Modular SOS semantics. Basic notions of structural operational semantics are assumed.

## 1 Introduction

Modular Structural Operational Semantics [17] (MSOS) is a variant of the Structural Operational Semantics framework that allows for the modular specification of the operational semantics of programming languages and concurrent systems.

Peter Mosses began the development of MSOS aiming at the development of a modular foundation for Action Semantics (AS) [16], challenged by Keith Wansbrough and John Hamer's work on a modular monadic semantics for Action Semantics [21]. Should Peter return to his Denotational Semantics roots [15] or insist on the operational semantics approach for AS foundation? The answer to this question was MSOS, a simple and yet powerful framework to support modular operational semantics specifications.

The development of MSOS began back in 1998 when Peter was visiting José Meseguer at SRI International while on sabbatical leave from Basic Research in Computer Science, Research Centre and International PhD School at the Universities of Aarhus and Aalborg (BRICS) and the Computer Science Department at the University of Aarhus. I was a PhD student then under the supervision of Edward Hermann Hæusler, a collaborator of Peter's since 1994. Thanks to

---

\* This research was sponsored by Ramn y Cajal program of the Ministerio de Ciencia y Innovacin de Espaa, project PROMESAS (S-0505/TIC/0407) from the Comunidad Autnoma de Madrid and PROPPi from Universidade Federal Fluminense.

a research project funded by NSF and the Brazilian Research Council, coordinated by José Meseguer, Armando Haeberer and later by Hermann himself, I became an international fellow at SRI working with three “beautiful minds”. I had the pleasure to observe how a framework is grown and to contribute to the research on MSOS using rewriting logic as a semantic framework for MSOS and the Maude system as a formal tool to support the specification and analysis of MSOS specifications [3,1,4,13,5,7,8,6,9].

In the constructive approach to programming language semantics [14], or Constructive Semantics for short, Peter proposes a library of basic abstract syntax (BAS) for language constructs commonly found in programming languages, such as a conditional, a loop or a function application. The semantics of a programming language may be expressed simply by relating its syntax with BAS constructs. The semantics of BAS constructs may be expressed in different semantic frameworks as long as there is support for modular specifications. MSOS is, obviously, one such framework. Action Semantics is another possibility. A modular monadic semantics is potentially yet another possibility but, to the best of my knowledge, there is no such formalization by the time this paper was written. Fabricio Chalub, in his Master’s dissertation, developed tool support, using the Maude system, for both MSOS and Constructive Semantics using MSOS as underlying framework [7]. I had the pleasure to advise his Master’s dissertation.

This paper reports on a Constructive Semantics for aspect-oriented programming (AOP) [12] constructs. AOP is a fairly recent programming discipline (around eleven years old). However, it gained worldwide acceptance due to its powerful programming model which essentially exposes metalevel facilities at the object level. The AOP approach, which was born at the Palo Alto Research Center (PARC)<sup>1</sup>, builds on Gregor Kiczales’ experience on metaobject protocols [10]. Kiczales’ AspectJ [11], an AOP extension to Java, is the *de facto* standard language for AOP. Today, AspectJ is an Eclipse project<sup>2</sup>.

The AOP approach defines a pointcut language and a weaving process. The **pointcut language** defines which are the points in the execution of a program (called join points in AOP terminology) which may be *intercepted* and *changed*. The join point model, that is, the points that may be intercepted during the execution of a program, is an essential component of a pointcut language’s semantics. Example join point are a method call and raising an exception. The change occurs using the so called *advices*. They may be executed, for instance, in three moments: *before* a join point, *replacing* its behavior (or around the join point, in AspectJ terminology) or *after* its execution. The **weaving process** introduces necessary code to merge an aspect and the code it intercepts. In AspectJ, this is done at bytecode level.

Recently, while exposed to AOP technology [2], and as an enthusiast of Peter’s work, I thought about how aspect-oriented programming constructs could be specified using Constructive Semantics with MSOS. The MSOS Constructive

---

<sup>1</sup> <http://www.parc.com/research/projects/aspectj/default.html>

<sup>2</sup> <http://www.eclipse.org/aspectj/>

Semantics for AOP constructs extends the MSOS semantics for BAS. I have specified BAS constructs to represent pointcut declarations, advices and join points and given a MSOS semantics to each one of them using MSOS rules. In this semantics, *join points* are identified with *small-step computations*. The *weaving* process is represented as a simple MSOS *rule order*. As an example, consider the specification for function application. (Please note that a particular function application *is a join point*.) In the MSOS Constructive Semantics for BAS there exists a MSOS rule to specify the *interception* of a functional application. In the Constructive Semantics for AOP constructs there is also a MSOS rule for functional application that first checks if there exists in the environment a declared pointcut that matches that particular application. If that is the case, the advice associated with the pointcut is evaluated. The point is that there can *not* be a *non-deterministic* choice between these two MSOS rules (the one for normal application and the one that intercepts the application). The MSOS rule that intercepts the application should have *priority* with respect to the one for normal application. *The transition rule captures the weaving process in a simple and yet powerful way.*

This paper is organized as follows. Section 2 gives an example of an aspect program in a very simple functional AOP language. Section 3 describes MSOS with the notion of order between transition rules and the constructs of Constructive Semantics necessary to our specification. Section 4 describes the main aspects of the Constructive Semantics of basic aspect constructs. Section 5 concludes the paper with some final remarks. The complete Constructive Semantics for AOP constructs is given in Appendix A.

## 2 A Simple Aspect-Oriented Program

The main constructs of an AOP language are *pointcut* and *advice* declarations. Essentially, a pointcut declares *which* join points will be intercepted and the advices state *what* will be done when the interception occurs.

Even though AspectJ is the best known AOP language nowadays, it does not mean that an AOP language has to be object-oriented. Example 1 is a program written in a ML-like functional language extended with a *pointcut* construction that declares a pointcut and associates a before, around and after advices, respectively, with it.

Before I explain the example, allow me first to informally recall the meaning of the ML syntax and basic AOP constructs used in this example.

In ML, the `let d in e end` expression evaluates expression *e* in the context of the declaration *d*. A declaration is written using syntax `val id = e` which binds the result of the evaluation of expression *e* to the identifier *id* in the environment. Possible *bindable values* are locations and closures. A *location* is produced when the expression `ref v` is evaluated with value *v*. At some point in the evaluation of the declaration, an identifier is bound to a location *l*, produced by the evaluation of `ref v`, which is bound to the value *v*. This produces an effect similar to pointers in imperative languages. A *closure*

is produced out of an abstraction evaluation. When the expression `fn id => e` is evaluated, it produces a bindable value the encloses the current environment (when static cope is considered, as in this case) together with the abstraction parameters, local variables and the expression that represents the abstraction body.

Regarding the basic AOP constructs used in the example, let me begin with the `pointcut(jp, ab, aa, aaf)` declaration. A pointcut declares the abstractions that will be executed before (`(ab)`), instead of (`(aa)`, or “around”, using AspectJ terminology, and after (`(aaf)`) the interception of a given join point (`(jp)`). The join point that represents the application of an abstraction is declared using syntax `call(id, v)`, which means the application of the abstraction bound to the identifier `id` to the value bound to `v`. Finally, the expression `proceed` returns the control flow to the point where the last join point was intercepted.

Of course, the syntax for these constructs used here is quite simplified when compared with a full-fledged language: for instance, we do not allow more than one parameter in an abstraction and the join point declaration does not allow a general pattern. These simplifications are, of course, for the sake of simplicity of presentation of my ideas, since the semantics of either feature is standard and explaining them would not contribute to the main point of this paper.

Let us move forward to Example 1. First, a variable `x` is declared and the value 1 is assigned to it. Functions `f`, `g` and `h` are declared and bound to abstractions that, respectively: (i) updates the value of the variable `x` with the value bound to `y`, (ii) `proceeds` and (iii) returns the value bound to the variable `x`. Next, a pointcut is declared for the application of `f` to `v`, with before, around and after advices given by functions `f`, `g` and `h` respectively. Finally, the function `f` is applied to the value bound the variable `x` within an environment defined by all the previous declarations.

*Example 1.*

```
let val x = ref 1 and val f = fn y => x := !x + y and
           val g = fn => proceed and
           val h = fn => !x
in let val pc = pointcut(call(f, v), f, g, h)
   in f !x
   end
end
```

To conclude this section, let me informally explain the evaluation of the expression in Example 1. When `f` is applied to `x` the application is intercepted and the environment searched for a pointcut that matches the application. The pointcut pattern matches with variable `v` becoming bound to 1 and the advices are enacted. First, function `f` is applied to 1 which increases the value bound to the location bound to `x` by one unit. Then, function `g` is applied to 1 and the command `proceed` is executed. The execution of `proceed` applies `f` to 1 once again and then increases the value bound to the location bound to `x` by one unit.

Finally, function  $h$  is applied to 1 and the value bound to the location bound to  $x$  is returned. The `let` expression returns 3, as function  $f$  is applied twice to 1.

## 3 The Semantic Framework

### 3.1 Modularity and MSOS

Modular SOS allows for the description of *modular* structural operational semantics specifications. The notion of modularity that MSOS supports is that constructs in a programming language may be specified *once and for all*. This means that an extension to the semantics of a given programming language with new features (that is, constructs and/or semantic components) do *not* require change on an existing specification of a given construct. A simple example of such a change is when an existing specification for expressions that manipulate an environment (such as declarations) is extended with commands that require a memory store (such as assignment). The need for a new semantic component, a store in this case, requires change to the existing specification of expression constructs: a new variable, representing the memory store, is added to each rule configuration. Notice that, in this case, the old constructs do not need to interact with the new semantic component. The point is that one would like to *conservatively* add new semantic components and constructs to existing specifications! So, how may the semantic framework give support for such specifications?

The support for modularity given by the MSOS framework lies on a simple and yet powerful twofold perception: (i) the configurations of a MSOS rule is allowed to have only syntax and computed values and (ii) the labels in the transitions are organized as indexed components in a record-like extensible structure.

Label components may be organized into three different disjoint sets: read-only, write-only or read-write. Read-only components refer to semantic entities such as the environment. Write-only may represent output or synchronization signals. Read-write are used to represent semantic entities such as the memory store. “Primed” indices represent the state of the components *after* the transition. Quoting [17]: *“Intuitively, a configuration in MSOS represents the part of the program which remains to be executed (and records already-computed values that are still needed), whereas the label on a transition represents all the information processed by the program in that step. Part of the label corresponds to the state of the processed information at the beginning of the transition, and part to its state at the end of the transition. For labels on adjacent transitions in a computation, the state at the end of the first transition must be identical to the state at the beginning of the second transition.”*

A concrete example may be helpful at this point. Let us consider the following specification for local expressions,

$$\begin{aligned} dec, dec' &\in Dec \\ dec &::= local(dec_1, dec_2) \\ \rho, \rho_0 &\in Env \quad exp, exp' \in Exp \end{aligned}$$

$$A = \{env : Env, \dots\}$$

$$\begin{array}{c} \frac{dec \xrightarrow{X} dec'}{local(dec, exp) \xrightarrow{X} local(dec', exp)} lcl-1 \\ \\ \frac{\begin{array}{c} exp \xrightarrow{\{env=\rho/\rho_0, \dots\}} exp' \\ local(\rho, exp) \xrightarrow{\{env=\rho_0, \dots\}} local(\rho, exp') \end{array}}{local(\rho, v) \longrightarrow v} lcl-2 \\ \\ lcl-3 \end{array}$$

where  $Dec$  is the set of declarations,  $local$  is the language construct for local declarations,  $Env$  is the set of environments,  $Exp$  is the set of expressions and  $A$  is the specification for the label structure.

Essentially, what rules  $lcl$  1 to 3 specify is that, to evaluate a local expression, first its declarations should be fully evaluated until they become an environment. Only then the expression argument of the local expression is evaluated in the context of the that environment.

Regarding labels, in rule  $lcl$ -1, the labels both in the conclusion and in the premise are denoted by the meta-variable  $X$ . This notation represents that label components may change in the transition. On the contrary, in rule  $lcl$ -3, the transition is considered to be *unobservable* as no change may occur in the label syntactically represented by a plain arrow with no variable what so ever. Finally, in rule  $lcl$ -2, the label is specified using a label pattern, as opposed to rule  $lcl$ -1, and the read-only label index  $env$ , which represents the current environment, is declared and bound to the component  $\rho_0$  in the conclusion. In its premise, the environment  $\rho_0$  is overridden<sup>3</sup> by the declarations in  $\rho$ .

This very simple example serves us well, as one may see: (i) the label structure is declared as an *extensible* record – this is the meaning of the  $\dots$  in the label declaration, (ii) the configurations in the rules either have syntax ( $local$  and its unevaluated arguments) or computed values (environment or the evaluated expression) and (iii) the rules only refer to the  $env$ -component, the one relevant for the semantics of the  $local$  construct.

It should be easy to see now that a specification for assignment would follow a similar *style* and its rules would only refer to a memory store. Thus, it would not be necessary to change this specification for local expressions when assignments

<sup>3</sup> The datatype that represents the environment is a finite map. Given two finite maps, the overwriting sum operation ( $\cup/-$ ) preserves the difference of the maps and substitutes the values of the first map in the intersection of the maps with the values of the second map.

are considered. Moreover, both this specification for local expressions and the one for assignments may be *reused* by other programming language semantics specifications that require such constructs.

### 3.2 Strategies in MSOS Specifications

In [6] Verdejo and I proposed a tool to support the specification of MSOS rule strategies in MSOS specifications. A rule strategy is a function that orders the rules by their labels. (The labels here are tags that uniquely identify a rule such as *lcl-1* in Section 3.1.) Therefore, when the rules are to be applied, the order defined by the given function must be used. We also discussed how Ordered SOS [19] (a SOS dialect that allows rules to be ordered) and negative premises could be represented in our setting.

The simplest strategies are the constants `idle`, which always succeeds by doing nothing, and `fail`, which always fails. The basic strategies consist of the application of a rule (identified by the corresponding rule label) to a given term, and with the possibility of providing a substitution for the variables in the rule. Some strategy combinators are concatenation (`;`), union (`|`) and the unary `not` that fails when its argument is successful and vice-versa. The strategy language in [6] is much more expressive than this subset but this should be enough for the purpose of this paper.

As an example, let us consider a priority operator  $\theta$  [18] from the CCS process algebra, which given a process  $P$  builds a new process that performs action  $\alpha$  or  $P$  if  $P$  cannot perform any action with a priority higher than  $\alpha$ . This operator is specified by the following rule scheme  $r_\theta$ .

$$\frac{X \xrightarrow{\alpha} X' \quad \forall_{\beta > \alpha} X \xrightarrow{\beta} r_\theta}{\theta(X) \xrightarrow{\alpha} \theta(X')}$$

Given a *finite* set of actions, the above scheme can be represented by many rules like  $r_\theta$  but without the negative premise and with an ordering among them. An example strategy for  $r_\theta$  is  $s = \text{not}(r_{\theta_c}\{s\} \mid r_{\theta_b}\{s\}) ; r_{\theta_a}\{s\} \mid \text{not}(r_{\theta_c}\{s\}) ; r_{\theta_b}\{s\} \mid r_{\theta_c}\{s\}$ , given a set of action labels  $\{a, b, c\}$  with the ordering  $\{a < b, a < c, b < c\}$ , and the rules for the priority operator labeled  $r_{\theta_a}$ ,  $r_{\theta_b}$ , and  $r_{\theta_c}$ . In [6] we have shown how a MSOS specification for CCS with the priority operator can be executed with the above strategy.

In this paper, I adopt the notation  $r_i > r_j$ , where  $r_i$  and  $r_j$  are rule labels (that identify the rules, they are not MSOS labels), denoting that rule  $r_i$  has priority over rule  $r_j$  with respect to rule application.

In the context of the AOP semantics in this paper, as mentioned in the introductory section and further explained in Section 4, a rule order that gives priority to a rule over another, as in the  $\theta$  operator above, precisely captures the notion of weaving when the given two rules specify the possible behaviors associated with a join point. (Note that a join point is essentially a step in our MSOS semantics.) Since rule ordering is one particular case of the more general approach proposed in [6], the AOP semantics proposed in this paper can be specified, executed and analysed using the approach proposed there.

### 3.3 Constructive Semantics

In the constructive approach to programming language semantics, Peter proposes a library of basic abstract syntax (BAS) for common programming language constructs. This library, when given a proper semantics using a modular framework, works similarly to Action Semantics' facets, by encapsulating the behavior of these common constructs.

The process of giving a Constructive Semantics to the concrete (or abstract) syntax of a programming language construct is syntax-driven: one simply has to give a translation for the concrete syntax of the programming language constructs to BAS constructs. This process can be quite straight forward as translating an *if* concrete syntax construct to a *cond* construct in BAS or not so trivial as representing objects as closures as Fabricio Chalub did in [7].

In this paper, however, I do not give such a translation. What I do is *extend* the MSOS specification of BAS constructs with new constructs to support AOP. Therefore, the semantics of an aspect-oriented programming language, such as AspectJ, can be specified by a translation from its syntax to the constructs I propose in this paper together with an object representation such as the one done by Fabricio in his dissertation.

In this paper, I will extend what could be understood as the “functional facet” of the Constructive Semantics of BAS, that is, an environment and constructs that operate it. An environment is a finite map between identifiers and bindable values. The *bind* operator evaluates to a pair of identifier and bindable value.

$$\text{Env} = (\text{Id}, \text{Bindable})\text{Map}$$

$$\begin{array}{llll} \text{dec} \in \text{Dec} & \text{id} \in \text{Id} & \text{exp}, \text{exp}' \in \text{Exp} & \text{v} \in \text{Value} \\ \text{dec} ::= \text{bind}(\text{id}, \text{exp}) \end{array}$$

$$\frac{\text{exp} \xrightarrow{X} \text{exp}'}{\text{bind}(\text{id}, \text{exp}) \xrightarrow{X} \text{bind}(\text{id}, \text{exp}')} \text{ bind}$$

$$\frac{}{\text{bind}(\text{id}, \text{v}) \longrightarrow \text{id} \mapsto \text{v}} \text{ bind}$$

A *local* expression is specified precisely by the description in Section 3.1. A *closure* simply encapsulates the current environment together with a given abstraction, where *Abs* is the set of abstractions and the dash in the rule label means that the remainder of the label structure is not observable in this transition, that is, it may not change in this transition.

$$a \in \text{Abs}$$

$$\frac{}{\text{close}(a) \xrightarrow{\{\text{env}=\rho, -\}} \text{closure}(\rho, a)} \text{ close}$$

The *application* (represented by the *app* construct) of an expression to an argument first fully evaluates its expression parameter and arguments. Finally, the

application becomes a local expression with the appropriate environment. This is specified by the *app* rules below, where *Exp* is the set of expressions, *Arg* is the set of arguments, *Par* is the set of parameters, *Passable* the set of actual parameters and *abs* is an abstraction constructor.

Note that the first two rules are actually non-deterministic. The point is that there is no reason why to impose an order on the evaluation of the application parameters as long as they become an abstraction and a passable value.

Section 4 proposes constructs to give semantics to AOP languages based on the above constructs.

$$\exp, \exp' \in \text{Exp} \quad \arg, \arg' \in \text{Arg} \quad \text{par} \in \text{Par} \quad \text{passable} \in \text{Passable}$$

$$\text{bindable} \in \text{Bindable} \quad \text{id} \in \text{Id}$$

$$\exp ::= \text{app}(\exp, \arg) \quad \text{par} ::= \text{bind}(\text{id})$$

$$\frac{\exp \xrightarrow{X} \exp'}{\text{app}(\exp, \arg) \xrightarrow{X} \text{app}(\exp', \arg)} \quad \text{app}$$

$$\frac{\arg \xrightarrow{X} \arg'}{\text{app}(\exp, \arg) \xrightarrow{X} \text{app}(\exp, \arg')} \quad \text{app}$$

$$\frac{}{\text{app}(\text{bind}(\text{id}), \text{bindable}) \longrightarrow (\text{id} \mapsto \text{bindable})} \quad \text{app}$$

$$\frac{}{\text{app}(\text{abs}(\text{par}, \exp), \text{passable}) \longrightarrow \text{local}(\text{app}(\text{par}, \text{passable}), \exp)} \quad \text{app}$$

## 4 A Constructive Semantics for Basic Aspect Constructs

Essentially, a pointcut language must allow the declaration of *pointcuts* and *advices*. In this paper, I propose a semantics for pointcuts over the *application* join point and *before*, *around* and *after* advices. In this section I cover the most important parts of the semantics. The complete specification can be found in Appendix A.

An advice is represented as a triple of closures. Each closure represents *before*, *around* and *after* advices, respectively. When a *before* or *after* advice declaration is not present they are replaced, in the triple, by the *nop* command that returns no value and produces no side-effect. The absence of the *around* advice is represented in the triple as the *proceed* command.

$$\text{abs}_1, \text{abs}_2, \text{abs}_3 \in \text{Abs}, \text{jp} \in \text{JoinPoint}$$

$$\text{dec} \in \text{Dec}$$

$$\text{dec} ::= \dots \mid \text{pointcut}(\text{jp}, \text{abs}_1, \text{abs}_2, \text{abs}_3)$$

A pointcut declaration inserts a new kind of binding into the environment: it maps a join point into an advice. In this paper we consider only the function application join point, represented by the construct *call*.

$$\begin{array}{c} jp \in JoinPoint, e \in Exp, arg \in Arg \\ jp ::= call(e, arg) \end{array}$$

When an application is evaluated, first the environment is checked for a pointcut declaration that matches the application at hand. If that is the case, the associated advice is evaluated. Otherwise, the rule for normal, “unintercepted” behavior, is triggered. This process is simply and cleanly captured by the rules for application in Section 3.3, that specify normal application behavior and by the rule *ctrl-app* that specifies intercepted behavior for application.

$$\frac{\begin{array}{c} abs = close(e_1) \quad def(\rho[call(e_1, e_2)]) \\ \hline app(e_1, e_2) \end{array}}{\{env=\rho, ctxt=\kappa, ctxt'=((abs, e_2), \kappa), \dots\} \rightarrow \rho[call(e_1, e_2)]} ctrl-app$$

*ctrl-app > app*

In rule *ctrl-app* *abs* ranges over *Abs*, *e<sub>i</sub>* ranges over *Exp* (expressions) and *app* is the rule label for functional application. The order between the rules, specified by *ctrl-app > app*, establishes that the rule for intercepted behavior should be tried first.

The long label at the conclusion of rule *ctrl-app* has two components, categorized as read-only and read-write, and three indices, *env*, *ctxt* and *ctxt'*. Note that the read-write component has two indices, *ctxt* and *ctxt'*, representing the component before and after the transition. The *env*-component is a finite map and represents the declarations environment, as in Section 3.1. The (*ctxt*, *ctxt'*)-component is a stack and represents the call stack of *intercepted* join points.

Note that the rule *ctrl-app* “pushes” a closure containing the current expression and the expression it is being applied to into the *ctxt'* component projection. This is necessary because the *proceed* command, specified below, needs that information to change the control flow back to “normal” behavior. The behavior of the command *proceed* is given by the following rule,

$$\frac{}{proceed \xrightarrow{\{ctxt=((closure(d, abs), e), \kappa), ctxt'=\kappa, \dots\}} local(d, app(abs, e))} proceed$$

where *d* ranges over *Dec*, *abs* range over *Abs*, *e* range over *Exp* and *κ* over *Context*. Note that the evaluation of *proceed* occurs within the evaluation of an advice. Therefore, if there exists an *after* advice to be evaluated, it will be executed after the evaluation of the *proceed* command.

## 5 Final Remarks

*Related work* There is a large literature on AOP. However, on AOP semantics the work in [20] is perhaps the closest to our approach, where the authors propose a

monadic semantics for Aspect Sand Box, a simplified AOP language designed to exercise different AOP models. Even though monads are used to give semantics to AOP concepts, such as advice, their semantics is not modular as Wansbrough and Hamer did for Action Semantics. In [20], the meaning of procedures is aware of AOP concepts. On page 901, the authors say, and I quote:

In a procedure call, first the arguments are evaluated in the usual call-by-value monadic way. Then, instead of directly calling the procedure, we use *enter-join-point* to create a new join point and enter it, invoking the weaver to apply any relevant advice.

It is important to emphasize my claim that an existing Constructive Semantics *description* for a given programming language (for instance, one that considers procedures) would *need not be affected at all* when AOP features are added to it due to the modular specification of the underlying constructs. One would only need to provide a proper translation from the desired concrete syntax for the AOP constructs into the ones I propose here.

On the other hand, the underlying transition system would, of course, be affected. It should be clear that programs in the resulting language, after the incorporation of aspects, will behave as AOP programs and therefore will (potentially) have different computations with respect to their non-aspect versions.

*Future work.* The pointcut language I use in this paper has *call* as its single pointcut descriptor (PCD). However, I do not foresee major problems to specify the meaning of other PCD<sup>4</sup> such as *cflow*, for instance. It would require the definition of a function that searches, within the *ctx*-component projection, for the latest join point that matches a given pattern. It is true, however, that rule *ctrl-app* would require a more general premise to match a more expressive PCD language, such as done in [20] with the *match-pcd* function. Also, new rules would be required to intercept other join points such as throwing an exception, which can be done in the same way as I do for function application.

These remarks do invalidate the contribution of this paper which is a reusable semantics for basic AOP constructs, with a clear and clean semantics for them. For instance, the notion of weaving, which can be cumbersome to formalize, is captured in a quite simple and precise way by a rule strategy. It should be clear that what I present here form a basis to *modularly* evolve this semantics to cope with new PCD and join points.

*Paper acknowledgements.* I would like to thank Edward Hermann Hæusler for his comments on a draft of this paper and the anonymous referees for their constructive comments and suggestions.

*Acknowledgement to Peter.* Peter Mosses deserves my deepest gratitude. His work has positively influenced me in different ways. Not only from a technical

---

<sup>4</sup> It is worth mentioning at this point that, should more complex rule strategies appear to be necessary to specify other join points, the complete machinery of rule strategies is at one's disposal and not only rule ordering.

point of view but also by its meaningfulness and simplicity. Peter has the gift of knowing how to translate non-trivial concepts into clean, clear and simple terms. It has been a pleasure and an honor to work with him as a PhD student and to collaborate with him afterwards. I wish him many springs more to come.

## References

1. Braga, C.: Rewriting Logic as a Semantic Framework for Modular Structural Operational Semantics. PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro (September 2001), <http://www.ic.uff.br/~cbraga>
2. Braga, C.: From access control policies to an aspect-based infrastructure: A metamodel-based approach. In: Chaudron, M. (ed.) MODELS 2008. LNCS, vol. 5421, pp. 243–256. Springer, Heidelberg (2009)
3. Braga, C., Haeusler, E.H., Meseguer, J., Mosses, P.D.: Maude action tool: Using reflection to map action semantics to rewriting logic. In: Rus, T. (ed.) AMAST 2000. LNCS, vol. 1816, pp. 407–421. Springer, Heidelberg (2000), [http://dx.doi.org/10.1007/3-540-45499-3\\_29](http://dx.doi.org/10.1007/3-540-45499-3_29)
4. Braga, C., Haeusler, E.H., Meseguer, J., Mosses, P.D.: Mapping modular SOS to rewriting logic. In: Leuschel, M. (ed.) LOPSTR 2002. LNCS, vol. 2664, pp. 262–277. Springer, Heidelberg (2003), [http://dx.doi.org/10.1007/3-540-45013-0\\_21](http://dx.doi.org/10.1007/3-540-45013-0_21)
5. Braga, C., Meseguer, J.: Modular rewriting semantics in practice. Electronic Notes in Theoretical Computer Science 117, 393–416 (2005), <http://dx.doi.org/10.1016/j.entcs.2004.06.019>
6. Braga, C., Verdejo, A.: Modular SOS with strategies. Electronic Notes in Theoretical Computer Science 175(1), 3–17 (2006), <http://dx.doi.org/10.1016/j.entcs.2006.10.024>
7. Chalub, F.: An implementation of modular structural operational semantics in maude. Master's thesis, Computer Science Graduate Program, Universidade Federal Fluminense (2005)
8. Chalub, F., Braga, C.: Maude MSOS Tool. Electronic Notes in Theoretical Computer Science 176(4), 133–146 (2006), <http://dx.doi.org/10.1016/j.entcs.2007.06.012>
9. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C., Braga, C., Farzan, A., Hendrix, J., Olveczky, P., Palomino, M., Sasse, R., Stehr, M.-O., Verdejo, A.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350, pp. 667–693. Springer, Heidelberg (2007), [http://dx.doi.org/10.1007/978-3-540-71999-1\\_21](http://dx.doi.org/10.1007/978-3-540-71999-1_21)
10. Kiczales, G., des Rivières, J., Bobrow, D.G.: The art of metaobject protocol. MIT Press, Cambridge (1991)
11. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of aspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)
12. Kiczales, G., Lampert, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.-M., Irwin, J.: Aspect-oriented programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
13. Meseguer, J., Braga, C.: Modular rewriting semantics of programming languages. In: Ratnay, C., Maharaj, S., Shankland, C. (eds.) AMAST 2004. LNCS, vol. 3116, pp. 364–378. Springer, Heidelberg (2004), <http://dx.doi.org/10.1007/b98770>

14. Mosses, P.: A constructive approach to language definition. *Journal of Universal Computer Science* 11(7), 1117–1134 (2005),  
[http://www.jucs.org/jucs\\_11\\_7/a\\_constructive\\_approach\\_to](http://www.jucs.org/jucs_11_7/a_constructive_approach_to)
15. Mosses, P.D.: Denotational semantics. In: *Handbook of theoretical computer science: formal models and semantics*, vol. B, MIT Press, Cambridge (1990)
16. Mosses, P.D.: Action semantics. Cambridge University Press, New York (1992)
17. Mosses, P.D.: Modular structural operational semantics. *Journal of Logic and Algebraic Programming* 60–61, 195–228 (2004)
18. Mousavi, M.: Structuring Structural Operational Semantics. PhD thesis, Technische Universiteit Eindhoven (2005)
19. Ulidowski, I., Phillips, I.: Ordered SOS process languages for branching and eager bisimulation. *Inf. Comput.* 178(1), 180–213 (2002)
20. Wand, M., Kiczales, G., Dutchyn, C.: A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems* 26(5), 890–910 (2004)
21. Wansbrough, K.: A modular monadic action semantics. Master’s thesis, Department of Computer Science, University of Auckland (1997)

## A Formal Semantics

Note: This specification is based on the MSDF specification for BAS in  
<http://maude-msos-tool.sf.net/specs/cmsos>.

### A.1 Sets of Auxiliary Entities

**Environments** :  $\rho \in Env = Id \uplus JoinPoint \rightarrow_{fin} DVal$   
**Denotable values** :  $DVal = Loc \uplus Advice$   
 $\forall \rho \in Env, \rho : Id \rightarrow_{fin} Loc \vee \rho : JoinPoint \rightarrow_{fin} Advice$   
**Advice** :  $adv \in Advice = Abs \times Abs \times Abs$   
**Context** :  $\kappa \in Context = (Abs \times Passable)^*$

### A.2 Label Components

$$A = \{env : Env; ctxt, ctxt' : Context; \dots\}$$

### A.3 Declarations

#### Joinpoint declaration syntax

$$\begin{aligned} jp &\in JoinPoint, e \in Exp, arg \in Arg \\ jp &::= call(e, arg) \end{aligned}$$

#### Bind overloading declaration syntax

$$\begin{aligned} dec &\in Dec, jp \in JoinPoint \\ dec &::= \dots \mid bind(jp, adv) \end{aligned}$$

## Pointcut and advice declaration syntax

$$abs_1, abs_2, abs_3 \in Abs, jp \in JoinPoint$$

$$dec \in Dec$$

$$dec ::= \dots \mid pointcut(jp, abs_1, abs_2, abs_3)$$

## Pointcut declaration semantics (Cons/Dec/pointcut)

$$\frac{}{pointcut(call(e_1, e_2), abs_1, abs_2, abs_3) \longrightarrow bind(call(e_1, e_2), (close(abs_1), close(abs_2), close(abs_3))))} pc-dec$$

where  $e_i$  range over  $Exp$  (expressions) and  $abs_i$  range over  $Abs$  (abstractions).

## A.4 Expressions

### Proceed syntax

$$\begin{aligned} e \in Exp \\ e ::= \dots \mid proceed \end{aligned}$$

## Advice semantics (Cons/Exp/advice)

$$\frac{\begin{array}{c} app(closure(d_1, b), e_1) \xrightarrow{\{cxt=((abs, e_1), \kappa_1), cxt'=\kappa_2, \dots\}} e_2 \\ (closure(d_1, b), closure(d_2, ar), closure(d_3, af)) \xrightarrow{\{cxt=((abs, e_1), \kappa_1), cxt'=\kappa_2, \dots\}} (e_2, closure(d_2, ar), closure(d_3, af)) \end{array}}{(e, closure(d, ar), closure(d, af)) \xrightarrow{X} (e', closure(d, ar), closure(d, af))} adv-eval_1$$

$$\frac{e \xrightarrow{X} e'}{(e, closure(d, ar), closure(d, af)) \xrightarrow{X} (e', closure(d, ar), closure(d, af))} adv-eval_2$$

$$\frac{app(closure(d, ar), v) \xrightarrow{X} e}{(v, closure(d, ar), closure(d, af)) \xrightarrow{X} (v, e, closure(d, af))} adv-eval_3$$

$$\frac{e \xrightarrow{X} e'}{(v, e, closure(d, af)) \xrightarrow{X} (v, e', closure(d, af))} adv-eval_4$$

$$\frac{app(closure(d, af), v_2) \xrightarrow{X} e}{(v_1, v_2, closure(d, af)) \xrightarrow{X} (v_1, v_2, e)} adv-eval_5$$

$$\frac{e \xrightarrow{X} e'}{(v_1, v_2, e) \xrightarrow{X} (v_1, v_2, e')} \text{adv-eval}_6$$

$$\frac{}{(v_1, v_2, v_3) \longrightarrow v_3} \text{adv-eval}_7$$

where  $abs$  ranges over  $Abs$ ,  $d_i$  ranges over  $Dec$  (declarations),  $a, b, ar$  and  $af$  over  $Abs$  (abstractions),  $e_i$  and  $e'$  over  $Exp$ ,  $\kappa_i$  over  $Context$  and  $v_i$  over computed values.

### Proceed semantics (Cons/Exp/proceed)

$$\frac{}{proceed \xrightarrow{\{cxt=((closure(d,abs),e),\kappa),cxt'=\kappa,\dots\}} local(d, app(abs,e))} proceed$$

where  $d$  ranges over  $Dec$ ,  $abs$  range over  $Abs$ ,  $e$  range over  $Exp$  and  $\kappa$  over  $Context$ .

### A.5 Abstractions

#### Controlled applications semantics (Cons/Abs/ctrl-app)

$$\frac{abs = close(e_1) \quad def(\rho[call(e_1, e_2)])}{app(e_1, e_2) \xrightarrow{\{env=\rho,cxt=\kappa,cxt'=((abs,e_2),\kappa),\dots\}} \rho[call(e_1, e_2)]} ctrl-app$$

$ctrl-app > app$

where  $abs$  ranges over  $Abs$ ,  $e_i$  ranges over  $Exp$  (expressions) and  $\{app\}$  is the set of rule labels for functional application.

# Structural Operational Semantics for Weighted Transition Systems

Bartek Klin\*

Warsaw University, University of Cambridge

**Abstract.** Weighted transition systems are defined, parametrized by a commutative monoid of weights. These systems are further understood as coalgebras for functors of a specific form. A general rule format for the SOS specification of weighted systems is obtained via the coalgebraic approach of Turi and Plotkin. Previously known formats for labelled transition systems (GSOS) and stochastic systems (SGSOS) appear as special cases.

## 1 Introduction

In its simplest and most well-studied form [1], Structural Operational Semantics (SOS) is a framework for inductive definition of transition systems labeled with some entities that have no internal structure, such as channel names in the process algebra CCS [2]. A rich theory of such SOS specifications have been developed including, among many other results, rule formats such as GSOS [3]. These guarantee good properties of bisimilarity, the canonical notion of equivalence on labeled transition systems.

However, already from the original paper that introduced SOS [4,5] it is apparent that for all but the simplest applications, one needs to consider systems where transitions carry more information than simple, unstructured labels. To model various aspects of computation, one endows transitions with information on fresh and bound names [6], transition probabilities [7] or durations [8], memory states, environments and so on. Crucially, the additional structure put on transitions influences the corresponding notion of process equivalence, so the simple theory of SOS and bisimilarity cannot be directly applied to these extended specification frameworks.

An important advantage of studying the structure of labels and transitions is that in a description of a rule format, or in a concrete specification of a system, one can specify that structure only partially and leave inessential details for further stages of specification. This allows for modularity in operational specifications, where different parts of a language can be specified independently, and extending an already specified language requires only an instantiation of the partially specified structure of transitions and labels, rather than its redefinition. This idea was promoted by Peter Mosses in his Modular Structural Operational

---

\* This work was supported by EPSRC grant EP/F042337/1.

Semantics (MSOS, [9,10]), where transition labels are thought of as arrows in categories, while the structure of these categories models various relevant aspects of computation and can be partially abstracted away in concrete specifications. MSOS achieves excellent modularity of SOS specifications that involve environments of various sorts, memory stores and communication channels. As a theory of process equivalence for MSOS specifications is missing, it is less clear how to meaningfully apply the framework for some aspects of computation such as probability or time.

A more radical abstraction step is the approach of *universal coalgebra* [11]. There, the only assumption on the structure of transitions and labels is that it corresponds to an endofunctor on some category (most often, the category **Set** of sets and functions). In particular, the notions of a transition or label are abstracted away in the coalgebraic framework. This approach provides a useful theory of process equivalence based on coalgebraic bisimilarity. Moreover, in [12] a general coalgebra-based theory of SOS specifications was developed. However, that *abstract GSOS* theory is rather detached from the usual transition-and-label presentations of SOS specifications, and it typically takes a considerable effort to understand its instances for particular classes of systems (see [13,14,15]). Also, the theory of modularity for abstract GSOS specification does not exist, apart from some initial results of [16,17].

This paper attempts to be a modest step in between the two approaches: the coalgebraic framework is applied to a class of systems where one can meaningfully speak of transitions and labels exhibiting a certain structure. More specifically, we study *weighted transition systems*, where every labeled transition is associated with a weight drawn from a commutative monoid  $\mathfrak{W}$ . The monoid structure determines the way in which weights of alternative transitions combine. A uniform coalgebraic treatment of weighted transition systems is provided, including a concrete definition of  $\mathfrak{W}$ -weighted bisimulation, and a general well-behaved rule format called  $\mathfrak{W}$ -GSOS, parametrized by the underlying monoid of weights, is defined. Weighted transition systems generalize both ordinary LTS and stochastic transition systems of [15], as well as other potentially useful kinds of systems.

After some algebraic preliminaries in Section 2, in Section 3 weighted transition systems are defined on a concrete level, without any use of coalgebraic techniques. The abstract coalgebraic approach to processes and SOS is recalled in Section 4, and weighted transition systems are presented as coalgebras in Section 5. Section 6 contains the main technical result of this paper: a definition of the  $\mathfrak{W}$ -GSOS format. In Section 7, it is shown how  $\mathfrak{W}$ -GSOS instantiates to the previously known formats GSOS [3] and SGSOS [15], for specific choices of the monoid  $\mathfrak{W}$ .

In Sections 4–6, the notions of functor and natural transformation are used. For these and other basic categorical notions, consult the first chapters or any handbook of category theory; [18] is the classical reference.

## 2 Preliminaries

A *commutative monoid*  $\mathfrak{W} = (W, +, 0)$  is a set equipped with a binary, associative, commutative operation called *addition* with a unit called *zero*. The addition operation is extended in an obvious way to summation  $\sum$  on arbitrary finite (multi)sets of elements of  $W$  (in particular, the empty sum is defined to be 0).

Given a commutative monoid  $\mathfrak{W}$ , a function  $\beta : W^k \rightarrow W$  is *multiadditive* if for each  $i \in \{1, \dots, k\}$ :

$$\begin{aligned}\beta(w_1, \dots, w_{i-1}, 0, w_{i+1}, \dots, w_k) &= 0, \\ \beta(w_1, \dots, w_{i-1}, w_i + w'_i, w_{i+1}, \dots, w_k) &= \beta(w_1, \dots, w_{i-1}, w_i, w_{i+1}, \dots, w_k) \\ &\quad + \beta(w_1, \dots, w_{i-1}, w'_i, w_{i+1}, \dots, w_k).\end{aligned}$$

To save space when the  $w_i$  are given by similar expressions, we shall often write  $\beta(w_i)_{i=1..k}$  for  $\beta(w_1, \dots, w_k)$ .

A good source of multiadditive functions are *semirings* (without 1), i.e. structures  $(W, +, 0, \cdot)$  such that:

- $(W, +, 0)$  is a commutative monoid,
- $(W, \cdot)$  is a semigroup (i.e.,  $\cdot$  is associative but not necessarily commutative),
- $\cdot$  distributes over  $+$ :
  - $w \cdot (u + v) = (w \cdot u) + (w \cdot v)$
  - $(u + v) \cdot w = (u \cdot w) + (v \cdot w)$ ,
- 0 annihilates  $\cdot$ :
  - $0 \cdot w = w \cdot 0 = 0$ .

Indeed in a semiring, for any  $v \in W$ , the function  $\beta_v : W^k \rightarrow W$  defined by:

$$\beta_v(w_i)_{i=1..k} = v \cdot \prod_{i=1}^k w_i$$

is multiadditive, where  $\prod$  is the obvious extension of  $\cdot$  to finite sequences of elements of  $W$ .

For any set  $X$ , a function  $f : X \rightarrow W$  is *finitely supported* if  $f(x) \neq 0$  for only finitely many  $x \in X$ . Such functions can be extended to arbitrary subsets: for any  $C \subseteq X$ , define

$$f(C) = \sum_{x \in C} f(x).$$

The sum is well defined if  $f$  is finitely supported. This notation extends to multi-argument functions in an obvious way, i.e.,  $g(x, C) = \sum_{y \in C} g(x, y)$  etc.

We shall use standard terminology and notation related to algebraic signatures, terms and substitutions. A *signature*  $\Sigma$  is a set of operation symbols (also denoted  $\Sigma$ ) with an arity function  $ar : \Sigma \rightarrow \mathbb{N}$ . The set of  $\Sigma$ -terms with variables from a set  $X$  is denoted  $T_\Sigma X$ ; in particular,  $T_\Sigma \emptyset$  denotes the set of closed terms. For a function  $\sigma : X \rightarrow Y$ ,  $\sigma[-] : T_\Sigma X \rightarrow T_\Sigma Y$  denotes its extension to terms, defined by variable substitution.

### 3 Weighted Transition Systems

Consider any commutative monoid  $\mathfrak{W} = (W, 0, +)$ . Elements of  $W$  will be called *weights* and denoted  $v, w, \dots$

**Definition 1.** A  $\mathfrak{W}$ -weighted labelled transition system ( $\mathfrak{W}$ -LTS in short) is a triple  $(X, A, \rho)$  where

- $X$  is a set of *states* (or *processes*),
- $A$  is a set of *labels*,
- $\rho : X \times A \times X \rightarrow W$  is called the *weight function*.

To support intuitions based on classical labelled transition systems (LTSs), we shall write  $\rho(x \xrightarrow{a} y)$  for  $\rho(x, a, y)$ , and to say that  $\rho(x \xrightarrow{a} y) = w$  we shall write  $x \xrightarrow{a,w} y$ .

The latter notational convention suggests that weights can be understood as parts of labels. Indeed,  $\mathfrak{W}$ -LTSs labelled with  $A$  can be seen as ordinary LTSs labelled with  $A \times W$ , subject to a weight determinacy condition: for each  $x, y \in X$  and  $a \in A$ , there is exactly one  $w \in W$  for which  $x \xrightarrow{a,w} y$ .

**Definition 2.** A  $\mathfrak{W}$ -LTSs  $(X, A, \rho)$  is *image finite* if for each  $x \in X$  and  $a \in A$ , the set of  $y \in X$  such that  $\rho(x \xrightarrow{a} y) \neq 0$  is finite.

In the following, we will restrict attention to image finite  $\mathfrak{W}$ -LTSs only.

In the definition of a  $\mathfrak{W}$ -LTSs, the monoid structure of  $\mathfrak{W}$  was not used in any way. It is, however, crucial in the definition of *weighted bisimulation*:

**Definition 3.** Given a  $\mathfrak{W}$ -LTS  $(X, A, \rho)$ , a  $\mathfrak{W}$ -*bisimulation* is an equivalence relation  $R$  on  $X$  such that for each  $x, x' \in X$ ,  $xRx'$  implies that for each  $a \in A$  and each equivalence class  $C$  of  $R$ :

$$\sum_{y \in C} \rho(x, a, y) = \sum_{y \in C} \rho(x', a, y).$$

Processes  $x, x' \in X$  are  $\mathfrak{W}$ -*bisimilar* if they are related by some  $\mathfrak{W}$ -bisimulation.

Note how the commutative monoid structure of  $\mathfrak{W}$ , together with the image finiteness assumption, ensures that the weights above are well-defined.

It is straightforward to see that  $\mathfrak{W}$ -weighted bisimulations are closed under (transitive closures of) arbitrary unions, hence  $\mathfrak{W}$ -bisimilarity on any LTS is the largest  $\mathfrak{W}$ -bisimulation on it.

*Example 1.* Consider the monoid of logical values  $z = \{\text{ff}, \text{tt}\}$ , with logical disjunction as  $+$  and  $\text{ff}$  as the zero element.  $z$ -LTSs are exactly ordinary (image-finite) LTSs, and  $z$ -bisimulations are classical bisimulations (more precisely, bisimulation equivalences).

*Example 2.* For  $\mathbb{R}_0^+$  the monoid of nonnegative real numbers under addition,  $\mathbb{R}_0^+$ -LTSs are exactly *rated transition systems* used in [15] to model stochastic systems, and  $\mathbb{R}_0^+$ -bisimilarity is stochastic bisimilarity [15], called strong equivalence in [19].

*Example 3.* The set  $\mathbb{R}^{+\infty}$  of positive real numbers augmented with positive infinity  $\infty$ , forms a commutative monoid with the minimum operation as addition and  $\infty$  as the zero element.  $\mathbb{R}^{+\infty}$ -LTSs themselves are almost the same as  $\mathbb{R}_0^+$ -LTSs of Example 2, with the only difference in the capability of making transitions with weight 0 or  $\infty$ . However, the different monoid structures lead to different notions of weighted bisimilarity and, as a result, to very different intuitions about the roles of weights in these systems. Indeed, while in Example 2 rates model the *capability* of a process to make a transition, with the idea that two similar capabilities add up to a stronger one, here weights might correspond to the *cost* of transitions, with the intuition that out of several similar possibilities, a process will always choose that of the lowest cost.

## 4 Abstract GSOS

In [12] (see [20] for a more elementary introduction), Turi and Plotkin proposed an abstract way of understanding well-behaved structural operational semantics for systems of various kinds. There, behaviour of transition systems is modeled by coalgebras, and their syntax by algebras. For example, image-finite LTSs labelled with elements of  $A$  can be understood as functions  $h : X \rightarrow (\mathcal{P}_\omega X)^A$ , where  $\mathcal{P}_\omega$  is the finite powerset construction. More generally, for any covariant functor  $B$  on the category **Set** of sets and functions, a  $B$ -coalgebra is a set  $X$  endowed with a function  $h : X \rightarrow BX$ . A  $B$ -coalgebra *morphism* from a  $h : X \rightarrow BX$  to  $g : Y \rightarrow BY$  is a function  $f : X \rightarrow Y$  such that  $g \circ f = Bf \circ h$ . The kernel relations of coalgebra morphisms are called *cocongruences* on their domains. Processes  $x, y \in X$  are *observationally equivalent* with respect to  $h : X \rightarrow BX$  if they are related by a cocongruence on  $h$ . For more information about the coalgebraic approach to process theory, see [11].

More traditionally, process syntax is modeled via algebras for endofunctors. Every algebraic signature  $\Sigma$  corresponds to a functor  $\Sigma X = \coprod_{\mathbf{f} \in \Sigma} X^{ar(\mathbf{f})}$  on **Set**, in the sense that a model for the signature is exactly an *algebra* for the functor, i.e., a set  $X$  and a function  $g : \Sigma X \rightarrow X$ . The set of  $\Sigma$ -terms with variables from a set  $X$  is denoted  $T_\Sigma X$ . In particular,  $T_\Sigma \emptyset$  is the set of closed terms over  $\Sigma$ ; it admits an obvious algebra structure  $a : \Sigma T_\Sigma \emptyset \rightarrow T_\Sigma \emptyset$  for the functor corresponding to the signature. This is the *initial*  $\Sigma$ -algebra. The construction  $T_\Sigma$  is also a functor, called the *free monad* over  $\Sigma$ .

In [12], Turi and Plotkin observed (a full proof was provided later by Bartels [13]), that operational LTS specifications in the well-known image finite GSOS format [3] are in an essentially one-to-one correspondence with *distributive laws*, i.e., natural transformations of the type

$$\lambda : \Sigma(\text{Id} \times B) \Longrightarrow BT_\Sigma \quad (1)$$

where  $B = (\mathcal{P}_\omega -)^A$  is the behaviour functor used for modeling LTSs,  $\Sigma$  is the functor corresponding to the given signature, and  $T_\Sigma$  is the free monad over  $\Sigma$ . Moreover, any  $\lambda$  as above gives rise to a  $B$ -coalgebra structure  $h_\lambda$  on  $T_\Sigma \emptyset$ ,

defined by a “structural recursion theorem” (see [12] for details) as the only function  $h_\lambda : T_\Sigma 0 \rightarrow BT_\Sigma 0$  such that:

$$h_\lambda \circ a = Ba^\sharp \circ \lambda_X \circ \Sigma\langle \text{id}, h_\lambda \rangle, \quad (2)$$

where  $a^\sharp : T_\Sigma T_\Sigma \emptyset \rightarrow T_\Sigma \emptyset$  is the inductive extension of  $a$ .

The fact that bisimilarity on LTSs induced from GSOS specifications is guaranteed to be a congruence, can be proved at the level of coalgebras and distributive laws:

**Theorem 1 ([12], Cor. 7.5).** Assume  $B$  has a final coalgebra. For any  $\lambda$  as in (1), observational equivalence on  $h_\lambda : T_\Sigma \emptyset \rightarrow BT_\Sigma \emptyset$  is a congruence on  $T_\Sigma \emptyset$ .

Based on this result, the search for congruence formats for weighted transition systems should begin from understanding them as coalgebras.

## 5 Weighted Transition Systems as Coalgebras

As before, we start with a commutative monoid  $\mathfrak{W} = (W, 0, +)$ . For any set  $X$ , a function  $\phi : X \rightarrow W$  is *finitely supported* if  $\phi(x) \neq 0$  for only finitely many  $x \in X$ . Let  $\mathcal{F}_{\mathfrak{W}} X$  denote the set of all finitely supported functions from  $X$  to  $W$ . This extends to an endofunctor  $\mathcal{F}_{\mathfrak{W}}$  on the category **Set** of sets and functions, with the action on functions defined by:

$$\mathcal{F}_{\mathfrak{W}} f(\phi)(y) = \sum_{x \in \overleftarrow{f}(y)} \phi(x)$$

for any  $f : X \rightarrow Y$ ,  $\phi \in \mathcal{F}_{\mathfrak{W}} X$  and  $y \in Y$  (here and in the following,  $\overleftarrow{f}(y) = \{x \in X \mid f(x) = y\}$ ). It is easy to see that  $\mathcal{F}_{\mathfrak{W}} f(\phi)$  is finitely supported if  $\phi$  is so, and that  $\mathcal{F}_{\mathfrak{W}}$  preserves identities and function composition.

**Proposition 1.** For any  $\mathfrak{W}$ , and any set  $A$ , coalgebras for the functor  $(\mathcal{F}_{\mathfrak{W}} -)^A$  are in one-to-one correspondence with  $\mathfrak{W}$ -LTSs labelled with  $A$ .

*Proof.* Any  $\mathfrak{W}$ -LTS  $(X, A, \rho)$  determines a coalgebra  $h : X \rightarrow (\mathcal{F}_{\mathfrak{W}} X)^A$  by  $h(x)(a)(y) = \rho(x \xrightarrow{a} y)$ . Image-finiteness of  $(X, A, \rho)$  means exactly that  $h(x)(a)$  is finitely supported. This correspondence is bijective.  $\square$

This coalgebraic understanding is justified by the corresponding treatment of weighted bisimilarity:

**Proposition 2.** For any  $\mathfrak{W}$ -LTS  $(X, A, \rho)$ , an equivalence relation on  $X$  is a  $\mathfrak{W}$ -bisimulation if and only if it is the kernel relation of a coalgebra morphism from the corresponding  $\mathcal{F}_{\mathfrak{W}}$ -coalgebra. As a corollary, two processes are  $\mathfrak{W}$ -bisimilar if and only if they are observationally equivalent.

*Proof.* See Appendix A.  $\square$

*Remark 1.* See Appendix B for an explanation as to why observational equivalence is used here instead of the approach of coalgebraic bisimilarity, based on spans of coalgebra morphisms.

To apply the general machinery of bialgebraic operational semantics, the following technical result is needed:

**Proposition 3.** The functor  $(\mathcal{F}_{\mathfrak{W}} -)^A$  admits a final coalgebra.

*Proof.* As proved in [21], it is enough to show that  $\mathcal{F}_{\mathfrak{W}}$  is finitary, i.e. that for any set  $X$  and any  $x \in \mathcal{F}_{\mathfrak{W}}X$  there is a finite subset  $Y \subseteq X$  such that  $x$  arises as an element of  $\mathcal{F}_{\mathfrak{W}}Y$ . But this easily follows from the assumption that  $\mathcal{F}_{\mathfrak{W}}X$  only contains finitely supported functions.  $\square$

## 6 Weighted GSOS

From Section 4 it follows that as well-behaved compositional specifications of  $\mathfrak{W}$ -LTSSs, one may take some syntactic entities (for example, sets of rules) that define natural transformations:

$$\lambda : \Sigma(\text{Id} \times (\mathcal{F}_{\mathfrak{W}} -)^A) \Longrightarrow (\mathcal{F}_{\mathfrak{W}} T_{\Sigma} -)^A \quad (3)$$

where  $\Sigma$  is the process syntax signature endofunctor and  $T_{\Sigma}$  is the free monad over  $\Sigma$ . Moreover, for any such syntactic entity, weighted bisimilarity is a congruence for the WTS obtained by from the corresponding  $\lambda$  by the inductive definition (2).

For  $\mathfrak{W} = \mathbb{Z}$  (see Example 1), image-finite GSOS specifications [3] define (3), as noticed in [12] and proved in [13]. Moreover, every natural transformation of type (3) arises from a GSOS specification. A similar full characterisation result, based on earlier developments of [13], was proved in [15] for  $\mathfrak{W} = \mathbb{R}_0^+$ , where a format called SGSOS (Stochastic GSOS) was proposed as a way to specify rated transition systems (see Example 2) well-behaved with respect to stochastic bisimilarity.

We shall now show a rule format, which we call  $\mathfrak{W}$ -GSOS, parametrized by a commutative monoid  $\mathfrak{W}$ , and see how specifications that conform to the format give rise to natural transformations of type (3). Both GSOS and SGSOS shall appear as special cases of this general format, as will be seen in Section 7.

Note, however, that we do not claim that every natural transformation of type (3) can be presented by a  $\mathfrak{W}$ -GSOS specification for every monoid  $\mathfrak{W}$ . In this sense, our results are weaker than those of [13] and [15], where full characterisation results for specific monoids were proved; nevertheless, we still provide a general congruence format for transition systems weighted by an arbitrary monoid.

In the following, fix an arbitrary commutative monoid  $\mathfrak{W} = (W, 0, +)$ .

**Definition 4 ( $\mathfrak{W}$ -GSOS rule).** A  $\mathfrak{W}$ -GSOS rule for a signature  $\Sigma$  and a set  $A$  of labels is an expression of the form:

$$\frac{\left\{ \mathbf{x}_i \xrightarrow{a} w_{a,i} \right\}_{a \in D_i, 1 \leq i \leq n} \quad \left\langle \mathbf{x}_{ij} \xrightarrow{b_j, u_j} \mathbf{y}_j \right\rangle_{1 \leq j \leq k}}{\mathbf{f}(\mathbf{x}_1, \dots, \mathbf{x}_n) \xrightarrow{c, \beta(u_1, \dots, u_k)} \mathbf{t}} \quad (4)$$

where

- $\mathbf{f} \in \Sigma$  and  $ar(\mathbf{f}) = n$ , with  $n, k \in \mathbb{N}$ , and  $\{i_1, \dots, i_k\} \subseteq \{1, \dots, n\}$ ;
- $\mathbf{x}_i$  and  $\mathbf{y}_j$  are all distinct variables taken from a fixed countably infinite set  $\Xi$ , and no other variables appear in  $\mathbf{t} \in T_\Sigma \Xi$ ;
- moreover, all variables  $\mathbf{y}_j$  appear in  $\mathbf{t}$ ;
- $D_i \subseteq A$ ;
- $w_{a,i} \in W$ ;
- $b_1, b_2, \dots, b_k, c \in A$ ;
- $u_1, \dots, u_k$  are pairwise distinct *weight variables* taken from a fixed countably infinite set  $\Upsilon$ ;
- $\beta : W^k \rightarrow W$  is a multiadditive function on  $\mathfrak{W}$ .

The set of variables from  $\Xi$  present in a rule  $R$  is denoted  $\Xi_R$ .

We now provide some terminology, notation and intuitions to aid the understanding of  $\mathfrak{W}$ -GSOS rules. The expression under the horizontal line in a rule is called the *conclusion*. The left side of the conclusion is called the *source* of a rule, and the right side is the *target*. Expressions above the horizontal line are *premises*. Each rule has premises of two kinds: *total weight* premises, depicted with  $\xrightarrow{a}$  arrows, and *transition* premises, where  $\xrightarrow{b}$  arrows are used.

Total weight premises form a set, i.e., their order in a rule is irrelevant. The set of total weight premises in a rule defines a partial function from  $\{1, \dots, n\} \times A$  to  $W$ , and the sets  $D_i$  describe its domain of definition for each  $i = 1..n$ . Intuitively, a total weight premise  $\mathbf{x} \xrightarrow{a} w$  is satisfied for a process  $x$  in a  $\mathfrak{W}$ -LTS, if the sum of weights of all  $a$ -labelled transitions from  $x$  equals  $w$ .

Transition premises in a rule form a sequence, i.e., their order is relevant. Note that the  $u_j$  in transition premises are not fixed weights (elements of  $W$ ), but variables. The meaning of a premise  $\mathbf{x} \xrightarrow{b, u} \mathbf{y}$  applied to a source process  $x$  and a target process  $y$  in a  $\mathfrak{W}$ -LTS is to assign the transition weight  $\rho(x \xrightarrow{b} y)$  to the variable  $u$ , used then as an argument in the function  $\beta$  mentioned in the rule conclusion. This process is formally described in Definition 6 below.

Weight variables in transition premises are somewhat redundant in a  $\mathfrak{W}$ -GSOS rule. Indeed, since each transition premise must come with a fresh weight variable, and the variables are then used only as arguments of  $\beta$  in an order prescribed by the order of premises, there is essentially only one way (up to renaming of weight variables) of putting them in any given rule. For brevity of notation, one can therefore omit weight variables altogether and write down  $\mathfrak{W}$ -GSOS rules as:

$$\frac{\left\{ \mathbf{x}_i \xrightarrow{a} w_{a,i} \right\}_{a \in D_i, 1 \leq i \leq n} \quad \left\langle \mathbf{x}_{ij} \xrightarrow{b_j} \mathbf{y}_j \right\rangle_{1 \leq j \leq k}}{\mathbf{f}(\mathbf{x}_1, \dots, \mathbf{x}_n) \xrightarrow{c, \beta} \mathbf{t}}$$

The former, full notation is useful as an intuitive reminder of where arguments of  $\beta$  come from; once one gets more familiar with the process of inducing  $\mathfrak{W}$ -LTSs from rules, the latter notation offers some welcome brevity.

We will be interested in collections of  $\mathfrak{W}$ -GSOS rules subject to a finiteness condition:

**Definition 5 ( $\mathfrak{W}$ -GSOS specification).** Given a signature  $\Sigma$  and a set  $A$  of labels, a  $\mathfrak{W}$ -GSOS specification  $\Lambda$  is a set of  $\mathfrak{W}$ -GSOS rules such that only finitely many rules share the same operator in the source ( $\mathbf{f}$ ), the same label in the conclusion ( $c$ ), and the same partial function from  $\{1, \dots, n\} \times A$  to  $W$  arising from their sets of total weight premises.

To complete the definition of  $\mathfrak{W}$ -GSOS, we must show how  $\mathfrak{W}$ -GSOS specifications induce  $\mathfrak{W}$ -LTSs.

**Definition 6 (induced  $\mathfrak{W}$ -LTS).** The  $\mathfrak{W}$ -LTS induced by a  $\mathfrak{W}$ -GSOS specification  $\Lambda$  over a signature  $\Sigma$  and a set of labels  $A$ , has the set  $T_{\Sigma}\emptyset$  of closed  $\Sigma$ -terms as states and  $A$  as the set of labels. The weight function  $\rho : T_{\Sigma}\emptyset \times A \times T_{\Sigma}\emptyset \rightarrow W$  is defined by structural induction on the first argument. To this end, consider a process  $s = \mathbf{f}(s_1, \dots, s_n) \in T_{\Sigma}\emptyset$  and assume that all  $\rho(s_i \xrightarrow{a} t)$  have been determined for all  $a \in A$  and  $t \in T_{\Sigma}\emptyset$ . For a fixed label  $c \in C$  and a process  $t \in T_{\Sigma}\emptyset$ , define  $\rho(s \xrightarrow{c} t)$  as follows.

We shall say that a rule  $R$  as in (4) *fits*  $s \xrightarrow{c} t$  if all of the following hold:

- (i) the operator in the source of  $R$  is  $\mathbf{f}$ ,
- (ii) the label in the conclusion of  $R$  is  $c$ ,
- (iii) for each total weight premise  $\mathbf{x}_i \xrightarrow{a} w$  in  $R$ , there is  $\rho(s_i, a, T_{\Sigma}\emptyset) = w$  (i.e. total weight premises are satisfied),
- (iv) there exists a substitution  $\sigma : \Xi_R \rightarrow T_{\Sigma}\emptyset$  such that:
  - $\sigma\mathbf{x}_i = s_i$  for  $i = 1, \dots, n$ , and
  - $\sigma[\mathbf{t}] = t$ .

It is important to note that if  $R$  fits  $s \xrightarrow{c} t$ , then the fitting substitution  $\sigma$  is unique. Indeed, the action of  $\sigma$  on the  $\mathbf{x}_i$  is explicitly defined by  $\sigma\mathbf{x}_i = s_i$ , and the action on the  $\mathbf{y}_j$  is determined by the condition  $\sigma[\mathbf{t}] = t$ . This is easily proved by structural induction on  $\mathbf{t}$ , using the assumption that all variables  $\mathbf{y}_j$  are present in  $\mathbf{t}$ .

If a rule  $R$  fits  $s \xrightarrow{c} t$ , its *contribution* to the weight of  $s \xrightarrow{c} t$  is a value in  $W$  calculated by:

$$\gamma(R) = \beta \langle \rho(s_{i_j}, b_j, \sigma\mathbf{y}_j) \rangle_{j=1..k}.$$

We then define  $\rho(s \xrightarrow{c} t)$  as the sum, taken in  $\mathfrak{W}$ , of contributions of all rules in  $\Lambda$  that fit  $s \xrightarrow{c} t$ . The sum exists thanks to the finiteness condition in Definition 5.

**Theorem 2.** Every  $\mathfrak{W}$ -GSOS specification  $\Lambda$  gives rise to a natural transformation  $\lambda$  as in (3). Moreover, the coalgebra induced from  $\lambda$  according to (2), coincides with the  $\mathfrak{W}$ -LTS induced from  $\Lambda$  according to Definition 6.

*Proof.* See Appendix C.

**Corollary 1.** For any  $\mathfrak{W}$ -GSOS specification  $\Lambda$ ,  $\mathfrak{W}$ -bisimilarity is a congruence on the  $\mathfrak{W}$ -LTS induced by  $\Lambda$ .

*Proof.* Use Theorems 1 and 2 with Propositions 2 and 3.

## 7 Examples

### 7.1 GSOS as $\mathfrak{z}$ -GSOS

To relate  $\mathfrak{W}$ -GSOS to a more familiar format, we shall now see what  $\mathfrak{W}$ -GSOS specifications look like for  $\mathfrak{W} = \mathfrak{z}$  (see Example 1).

First, there are only two kinds of total weight premises to consider: ones of the form  $x \xrightarrow{a} tt$  that require some  $a$ -transitions from a process corresponding to  $x$  to exist, and ones of the form  $x \xrightarrow{a} ff$ , that forbid such transitions. One can rewrite the former as  $x \xrightarrow{a} \triangleright$ , and the latter as  $x \xrightarrow{a} \not\triangleright$ .

Next, it is easy to see that for any  $k \in \mathbb{N}$ , there are only two multiadditive functions  $\beta : \{\text{tt}, \text{ff}\}^k \rightarrow \{\text{tt}, \text{ff}\}$  on the monoid  $\mathfrak{z}$ . Indeed, by multiadditivity axioms,  $\beta$  is fully determined by its value on the all-tt vector. Assigning  $\beta(\text{tt}, \dots, \text{tt}) = \text{tt}$  or  $\beta(\text{tt}, \dots, \text{tt}) = \text{ff}$  one obtains respectively the  $k$ -ary conjunction or the constantly  $\text{ff}$  function as  $\beta$ . It is easy to check that both are multiadditive. However, a rule with the constantly  $\text{ff}$  function as  $\beta$  cannot make a nonzero contribution to any transition in the induced  $\mathfrak{z}$ -LTS, therefore any  $\mathfrak{z}$ -GSOS specification does not change its meaning if all such rules are removed from it. One can therefore safely restrict attention to rules with logical conjunction as  $\beta$ ; this means that  $\beta$  can be left implicit in the description of each rule. Moreover, since conjunction is commutative, the order of transition premises in  $\mathfrak{z}$ -GSOS rules is irrelevant.

The above observations let one write  $\mathfrak{z}$ -GSOS rules in the form:

$$\frac{\left\{ x_i \xrightarrow{a} \triangleright \right\}_{a \in E_i, 1 \leq i \leq n} \quad \left\{ x_i \xrightarrow{a} \not\triangleright \right\}_{a \in B_i, 1 \leq i \leq n} \quad \left\{ x_i \xrightarrow{b_{ij}} y_{ij} \right\}_{1 \leq i \leq n, 1 \leq j \leq k_i}}{f(x_1, \dots, x_n) \xrightarrow{c} t} \quad (5)$$

where

- $f \in \Sigma$  and  $ar(f) = n$ , with  $n, k_i \in \mathbb{N}$ ;
- $x_i$  and  $y_{ij}$  are all distinct variables and no other variables appear in  $t \in T_\Sigma \Xi$ ; moreover, all variables  $y_{ij}$  appear in  $t$ ;
- $E_i, B_i \subseteq A$  and  $b_{ij}, c \in A$ .

The induction process described in Definition 6 specializes to the following procedure. For a process  $s = f(s_1, \dots, s_n) \in T_\Sigma$ , assume that all outgoing transitions from the  $s_i$  have been determined. For a fixed label  $c \in C$  and a process  $t \in T_\Sigma \emptyset$ , determine whether the transition  $s \xrightarrow{c} t$  is present, as follows.

A rule  $R$  as in (5) fits  $s \xrightarrow{c} t$  if all of the following hold:

- the operator in the source of  $R$  is  $f$ ,
- the label in the conclusion of  $R$  is  $c$ ,
- for each premise  $x_i \xrightarrow{a}$  in  $R$ , there is  $s_i \xrightarrow{a} u$  for some  $u$ , and for each premise  $x_i \xrightarrow{a}$  there is no transition  $s_i \xrightarrow{a} u$  for any process  $u$ ,
- there exists a substitution  $\sigma : \Xi_R \rightarrow T_\Sigma \emptyset$  such that:
  - $\sigma x_i = s_i$  for  $i = 1, \dots, n$ , and
  - $\sigma[t] = t$ .

If a rule  $R$  fits  $s \xrightarrow{c} t$ , it contributes the transition  $s \xrightarrow{c} t$  to the induced system if and only if for each premise  $x_i \xrightarrow{b_{ij}} y_{ij}$  in  $R$ , the transition  $s_i \xrightarrow{b_{ij}} \sigma y_{ij}$  is present. (The universal quantification in the previous sentence corresponds to the use of conjunction as  $\beta$ .) Then the transition  $s \xrightarrow{c} t$  is present in the induced system if any rule contributes it. (The existential quantification here corresponds to disjunction being the operator in the underlying monoid.)

It is clear that both the format (5) and the associated induction procedure are almost exactly those for the well-known GSOS format [3,1]; the only difference in rule presentation is than in GSOS, premises  $x \xrightarrow{a}$  are equipped with dummy target variables, and consequently the condition that all target variables of transition premises are present in  $t$ , is dropped. It is not difficult to see that this makes no semantic difference in this case.

## 7.2 SGSOS Rules as $\mathbb{R}_0^+$ -GSOS Rules

It is even easier to see that SGSOS specifications, defined in [15] as a way to specify Markovian transition systems, conform to the  $\mathbb{R}_0^+$ -GSOS format, for  $\mathbb{R}_0^+$  the monoid of nonnegative real numbers under addition (see Example 2). An SGSOS rule is an expression of the form:

$$\frac{\left\{ x_i \xrightarrow{a @ w_{ai}} \right\}_{a \in D_i, 1 \leq i \leq n} \left\{ x_{i,j} \xrightarrow{b_j} y_j \right\}_{1 \leq j \leq k}}{f(x_1, \dots, x_n) \xrightarrow{c, w} t}, \quad (6)$$

subject to conditions similar to that of  $\mathfrak{W}$ -GSOS (4); the only differences are the following:

- (i) total weight premises are denoted  $x \xrightarrow{a @ w}$  in SGSOS rather than  $x \xrightarrow{a} w$  in  $\mathbb{R}_0^+$ -GSOS,
- (ii) transition premises in SGSOS form a set rather than a sequence, i.e., their order is disregarded,
- (iii) in the SGSOS rule conclusion, a non-zero weight  $w \in \mathbb{R}^+$  is used rather than a multiadditive function  $\beta$ ,
- (iv) in SGSOS it is required that  $b_{ij} \in D_{i,j}$  for each  $j = 1, \dots, k$ ; in other words, each transition premise has a corresponding total weight premise, and moreover the corresponding total weight  $w_{b_{j,i_j}}$  is required to be non-zero.

The induction procedure of a stochastic transition system from an SGSOS specification is similar to that given in Definition 6. The notion of a fitting rule is exactly the same, but the contribution of a rule is defined a bit differently:

$$\gamma(R) = w \cdot \prod_{j=1}^k \frac{\rho(s_{ij} \xrightarrow{b_j} \sigma y_j)}{w_{b_j, i_j}}.$$

Any SGSOS rule, written down as in (6), can be encoded as a  $\mathbb{R}_0^+$ -GSOS rule:

$$\frac{\left\{ x_i \xrightarrow{a} w_{a,i} \right\}_{a \in D_i, 1 \leq i \leq n} \quad \left\langle x_{i_j} \xrightarrow{b_j} y_j \right\rangle_{1 \leq j \leq k}}{f(x_1, \dots, x_n) \xrightarrow{c, \beta} t} \quad (7)$$

with transition premises arranged in an arbitrary order, and with  $\beta : (\mathbb{R}_0^+)^k \rightarrow \mathbb{R}_0^+$  defined by:

$$\beta(w_1, \dots, w_k) = \frac{w}{\prod_{j=1}^k w_{b_j, i_j}} \cdot \prod_{j=1}^k w_j.$$

The numbers  $w_{b_j, i_j}$  are fixed, and the division is well-defined, by the above requirement (iv) on SGSOS rules. Since  $\beta$  is commutative, the order of transition premises can be chosen arbitrarily.

### 7.3 $\mathbb{R}^{+\infty}$ -GSOS

The appearance of  $\mathfrak{W}$ -GSOS rules does not depend on the monoid structure of  $\mathfrak{W}$  apart from the choice of functions  $\beta$ , so  $\mathbb{R}^{+\infty}$ -GSOS and  $\mathbb{R}_0^+$ -GSOS specifications look almost the same. Note that for the monoid  $\mathbb{R}^{+\infty}$  (see Example 3), where the minimum operation is taken as addition, a function  $\beta$  is multiadditive if and only if it is monotonic and preserves  $\infty$ , i.e.  $\beta(w_1, \dots, w_n) = \infty$  whenever some  $w_i = \infty$ . As a result,  $\mathbb{R}^{+\infty}$ -GSOS rules look as in (4), with the requirement that all  $w_{a,i} \in \mathbb{R}^{+\infty}$  and  $\beta$  is a monotonic,  $\infty$ -preserving function.

This rule format allows for SOS definition of several interesting operators aimed at compositional specification of cost-oriented transition systems. For example, a unary prefixing operator and a binary nondeterministic choice operator, with syntax given by:

$$P ::= \text{nil} \mid (a, w).P \mid P + P \quad (a \in A, w \in \mathbb{R}^+)$$

can be defined by rules:

$$\frac{}{(a, w).x \xrightarrow{a, w} x} \quad \frac{x \xrightarrow{a, u} x'}{x + y \xrightarrow{a, u} x'} \quad \frac{y \xrightarrow{a, u} y'}{x + y \xrightarrow{a, u} y'}$$

where in the first rule  $w$  represents the function constant at  $w$ , and in the other two rules the identity function is taken for  $\beta$ . By Definition 6 applied to  $\mathfrak{W} = \mathbb{R}^{+\infty}$ , contributions of different rules to single transitions are combined

using the minimum operation. As a result, for example, the process  $(a.2).\text{nil} + (a,3).\text{nil}$  is  $\mathbb{R}^{+\infty}$ -bisimilar to  $(a.2).\text{nil}$ , which corresponds to the intuition that nondeterministic processes always choose the lowest possible cost of transition.

Other versions of nondeterministic composition, where the process of resolving a nondeterministic choice is associated with its own internal cost, can also be modeled. For example, a binary operator  $_3+5$  is defined by rules:

$$\frac{x \xrightarrow{a,u} x'}{x_3+5 y \xrightarrow{a,u+3} x'} \quad \frac{y \xrightarrow{a,u} y'}{x_3+5 y \xrightarrow{a,u+5} y'}$$

Here, choosing the left summand of nondeterministic choice incurs a lower cost.

Various ways of synchronization are also possible. For example, one can add cost information to the well-known CCS communication rule for a binary parallel composition operator  $\parallel$ , as in:

$$\frac{x \xrightarrow{a,u} x' \quad y \xrightarrow{a,v} y'}{x \parallel y \xrightarrow{\tau,u+v} x' \parallel y'}$$

which can model a situation where two processes use a common resource during synchronization, so that their costs of single transitions are added together. Another option is:

$$\frac{x \xrightarrow{a,u} x' \quad y \xrightarrow{a,v} y'}{x \parallel y \xrightarrow{\tau,\max(u,v)} x' \parallel y'}$$

where, intuitively, the two processes do not compete for a shared resource. Any operation can be used for weight combination here, as long as it is monotonic and preserves  $\infty$ .

Additional flexibility is provided by total weight premises of  $\mathfrak{W}$ -GSOS, which can be used here to check the minimal weight among all transition originating in a given process. For example, for a weighted version of a priority operator, one might define a unary operator  $\partial_{ab}$  by taking rules:

$$\frac{x \xrightarrow{a} w \quad x \xrightarrow{b} v \quad x \xrightarrow{a,u} x'}{\partial_{ab}(x) \xrightarrow{a,u} \partial_{ab}(x')} \quad \frac{x \xrightarrow{a} v \quad x \xrightarrow{b} w \quad x \xrightarrow{b,u} x'}{\partial_{ab}(x) \xrightarrow{b,u} \partial_{ab}(x')}$$

for each  $w \leq v \in \mathbb{R}^{+\infty}$ . The resulting set of rules is uncountable, but it satisfies the finiteness condition of Definition 5. The operator defined by these rules preserves all  $a$ -labeled transitions if the minimal weight of an  $a$ -labeled outgoing transition is not bigger than the minimal weight of a  $b$ -labeled one, and vice versa.

All these operators conform to the  $\mathbb{R}^{+\infty}$ -GSOS format, so compositionality of  $\mathbb{R}^{+\infty}$ -weighted bisimilarity is immediately guaranteed for them.

## 8 Conclusions and Future Work

Several research directions are left open here. First of all, some interesting kinds of transition systems do not exactly fit in our framework, although they seem

quite close to it. For example, reactive probabilistic transition systems [7,22] are almost like  $\mathbb{R}_0^+$ -LTSs of Example 2, except for the requirement that for each process  $P$  and label  $a$ , weights of all  $a$ -labelled transitions from  $P$  add up to 1.

This motivates the study of *constrained* weighted transition systems, i.e., coalgebras for functors  $\mathcal{F}_{\mathfrak{W}}^V$  (where  $V \subseteq W$  is the *constraint*), defined on sets by:

$$\mathcal{F}_{\mathfrak{W}}^V X = \left\{ \phi \in \mathcal{F}_{\mathfrak{W}} X \mid \sum_{x \in X} \phi(x) \in V \right\}$$

and as  $\mathcal{F}_{\mathfrak{W}}$  on functions. For example, the probability distribution functor used in coalgebraic modeling of probabilistic systems is naturally isomorphic to  $\mathcal{F}_{\mathbb{R}_0^+}^{\{0,1\}}$ , and the subprobability distribution functor to  $\mathcal{F}_{\mathbb{R}_0^+}^{[0,1]}$ . Also various versions of deterministic systems can be modeled as coalgebras for suitable constrained weighted functors. However, a characterization of abstract GSOS natural transformations in terms of rule formats remains to be provided for bounded functors. We conjecture that the formats can be obtained by subjecting their unconstrained counterparts to additional constraint conditions on (collections of) multiadditive functions  $\beta$  used in rule conclusions.

Furthermore, contrary to previous developments on GSOS [13] and SGSOS [15] formats, we do not claim that  $\mathfrak{W}$ -GSOS fully characterizes abstract GSOS for  $\mathfrak{W}$ -LTSs, i.e. that every natural transformation as in (3) arises from a  $\mathfrak{W}$ -GSOS specification. We conjecture that this is not the case in general, and a characterization of those monoids  $\mathfrak{W}$  for which the full characterization does hold is currently missing.

Another promising topic of future work is the incorporation of commutative monoid morphisms as a means of inducing morphisms between weighted GSOS specifications. The general goal here is a modular framework for the specification of weighted systems, inspired by MSOS [9,10].

Last but not least, modal logics for reasoning about weighted systems should be developed along the lines of coalgebraic modal logic.

## References

1. Aceto, L., Fokkink, W.J., Verhoef, C.: Structural operational semantics. In: Bergstra, J.A., Ponse, A., Smolka, S. (eds.) *Handbook of Process Algebra*, pp. 197–292. Elsevier, Amsterdam (2002)
2. Milner, R.: *Communication and Concurrency*. Prentice-Hall, Englewood Cliffs (1988)
3. Bloom, B., Istrail, S., Meyer, A.: Bisimulation can't be traced. *Journal of the ACM* 42, 232–268 (1995)
4. Plotkin, G.D.: A structural approach to operational semantics. DAIMI Report FN-19, Computer Science Department, Aarhus University (1981)
5. Plotkin, G.D.: A structural approach to operational semantics. *Journal of Logic and Algebraic Programming* 60-61, 17–139 (2004)
6. Sangiorgi, D., Walker, D.: *The  $\pi$ -Calculus: a Theory of Mobile Processes*. Cambridge University Press, Cambridge (2003)
7. Larsen, K.G., Skou, A.: Bisimulation through probabilistic testing. *Information and Computation* 94, 1–28 (1991)

8. Moller, F., Tofts, C.: A temporal calculus of communicating systems. In: Baeten, J.C.M., Klop, J.W. (eds.) CONCUR 1990. LNCS, vol. 458, pp. 401–415. Springer, Heidelberg (1990)
9. Mosses, P.D.: Foundations of Modular SOS. In: Kutylowski, M., Wierzbicki, T., Pacholski, L. (eds.) MFCS 1999. LNCS, vol. 1672, pp. 70–80. Springer, Heidelberg (1999)
10. Mosses, P.D.: Modular structural operational semantics. Journal of Logic and Algebraic Programming 60–61, 195–228 (2004)
11. Rutten, J.J.M.M.: Universal coalgebra: a theory of systems. Theoretical Computer Science 249, 3–80 (2000)
12. Turi, D., Plotkin, G.D.: Towards a mathematical operational semantics. In: Proc. LICS 1997, pp. 280–291. IEEE Computer Society Press, Los Alamitos (1997)
13. Bartels, F.: On Generalised Coinduction and Probabilistic Specification Formats. PhD dissertation, CWI, Amsterdam (2004)
14. Kick, M.: Rule formats for timed processes. In: Proc. CMCIM 2002. ENTCS, vol. 68, pp. 12–31. Elsevier, Amsterdam (2002)
15. Klin, B., Sassone, V.: Structural operational semantic for stochastic systems. In: Amadio, R.M. (ed.) FOSSACS 2008. LNCS, vol. 4962, pp. 428–442. Springer, Heidelberg (2008)
16. Kick, M., Power, J., Simpson, A.: Coalgebraic semantics for timed processes. Information and Computation 204, 588–609 (2006)
17. Lenisa, M., Power, J., Watanabe, H.: Category theory for operational semantics. Theoretical Computer Science 327(1–2), 135–154 (2004)
18. Mac Lane, S.: Categories for the Working Mathematician, 2nd edn. Springer, Heidelberg (1998)
19. Hillston, J.: A Compositional Approach to Performance Modelling. Cambridge University Press, Cambridge (1996)
20. Klin, B.: Bialgebraic methods and modal logic in structural operational semantics. Information and Computation 207, 237–257 (2009)
21. Barr, M.: Terminal coalgebras in well-founded set theory. Theoretical Computer Science 114, 299–315 (1993)
22. de Vink, E.P., Rutten, J.J.M.M.: Bisimulation for probabilistic transition systems: A coalgebraic approach. Theoretical Computer Science 221(1–2), 271–293 (1999)
23. Moss, L.: Coalgebraic logic. Annals of Pure and Applied Logic 96, 177–317 (1999)

## A Proof of Proposition 2

For  $\mathfrak{W}$ -LTSs  $(X, A, \rho)$  and  $(Y, A, \theta)$ , it is easy to check that a function  $f : X \rightarrow Y$  is a morphism between the corresponding coalgebras if and only if, for each  $x \in X$ ,  $a \in A$  and  $y \in Y$ ,

$$\theta(f(x), a, y) = \rho(x, a, \overleftarrow{f}(y)).$$

We show that a relation  $R$  is a  $\mathfrak{W}$ -bisimulation if and only if it is a kernel relation of such a morphism.

In the “if” direction, assume  $x, x' \in X$  such that  $f(x) = f(x')$  for a coalgebra morphism  $f$ . Note that equivalence classes of the kernel relation  $\text{ker}(f)$  correspond bijectively to elements of  $Y$  in the image of  $f$ : for each equivalence class  $C$  there is a  $y \in Y$  such that  $C = \overleftarrow{f}(y)$ . This implies that

$$\rho(x, a, C) = \theta(f(x), a, y) = \theta(f(x'), a, y) = \rho(x', a, C)$$

hence  $\ker(f)$  is a  $\mathfrak{W}$ -bisimulation.

In the "only if" direction, given a  $\mathfrak{W}$ -bisimulation  $R$  on  $(X, A, \rho)$ , define a  $\mathfrak{W}$ -LTS on the set of  $R$ -equivalence classes  $(X/R, A, \theta)$  by:

$$\theta(C, a, C') = \rho(x, a, C');$$

this is well-defined since, by Definition 3,  $\rho(x, a, C') = \rho(y, a, C')$  for any  $x, y \in C$ . Furthermore, the quotient map  $[-]_R : X \rightarrow X/R$  is a coalgebra morphism, since  $\overleftarrow{[-]_R}(C) = C$ .  $\square$

## B $\mathcal{F}_{\mathfrak{W}}$ and Weak Pullback Preservation

When coalgebras for a functor  $B$  are considered, it is often assumed that  $B$  preserves weak pullbacks (see [11] for more details). Several useful results follow from this assumption. In particular, the notion of observational equivalence adopted in this paper coincides with another canonical notion, that of *coalgebraic bisimulation*, defined abstractly by means of spans of coalgebra morphisms. Also, the abstract GSOS machinery works for coalgebraic bisimulation only if the behaviour functor involved preserves weak pullbacks.

It turns out that our behaviour functors do not preserve weak pullbacks in general, therefore we choose to adopt the notion of observational equivalence, encouraged by Proposition 2.

We shall now see a characterization of those functors  $\mathcal{F}_{\mathfrak{W}}$  that do preserve weak pullbacks, in terms of the underlying monoids.

Inspired by Theorem 3.6 in [23], we say that a commutative monoid  $\mathfrak{W}$  has the *row-column property* if for every two vectors  $(w_i)_{i=1..n}, (v_j)_{j=1..m}$  of elements of  $W$  such that  $\sum_{i=1}^n w_i = \sum_{j=1}^m v_j$ , there exists a rectangular matrix  $(u_{ij})_{i=1..n, j=1..m}$  of elements of  $W$  such that  $\sum_{j=1}^m u_{ij} = w_i$  for each  $i = 1..n$  and  $\sum_{i=1}^n u_{ij} = v_j$  for each  $j = 1..m$ , as illustrated below:

$$\begin{array}{cccc|c} u_{11} & u_{12} & \cdots & u_{1m} & w_1 \\ u_{21} & u_{22} & \cdots & u_{2m} & w_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \hline u_{n1} & u_{n2} & \cdots & u_{nm} & w_n \\ \hline v_1 & v_2 & \cdots & v_m & \sum \end{array}$$

**Proposition 4.**  $\mathcal{F}_{\mathfrak{W}}$  preserves weak pullbacks if and only if  $\mathfrak{W}$  has the row-column property.

*Proof.* The reasoning in Example 3.5 in [23] works here without any essential change.  $\square$

All monoids used in examples throughout this paper have the row-column property. For a simple example of one that does not have it, consider the four-element monoid  $\{0, a, b, 1\}$ , with 0 as the zero element and with addition defined by:

$$\begin{array}{c|cccc} + & 0 & a & b & 1 \\ \hline 0 & 0 & a & b & 1 \\ a & a & 1 & 1 & 1 \\ b & b & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{array}$$

and check that the property fails for  $n = m = 2$  and  $w_1 = w_2 = a$ ,  $v_1 = v_2 = b$ .

## C Proof of Theorem 2

First we shall see how a single  $\mathfrak{W}$ -GSOS rule defines a natural transformation  $\lambda$  as in (3). For a  $\mathfrak{W}$ -GSOS rule  $R$ , and for any set  $X$ , consider an arbitrary  $s = \mathbf{f}(x_1, \delta_1, \dots, x_n, \delta_n) \in \Sigma(X \times (\mathcal{F}_{\mathfrak{W}} X)^A)$ . To define  $\lambda_X(s) \in (\mathcal{F}_{\mathfrak{W}} T_{\Sigma} X)^A$ , pick an arbitrary  $c \in A$  and  $t \in T_{\Sigma} X$  and define  $\lambda_X(s)(c)(t) \in W$  as follows.

Say that  $R$  fits  $s, c, t$  if:

- (i) the operator in the source of  $R$  is  $\mathbf{f}$ ,
- (ii) the label in the conclusion of  $R$  is  $c$ ,
- (iii) for each total weight premise  $\mathbf{x}_i \xrightarrow{a} w$  in  $R$ , there is  $\sum_{y \in X} \delta_i(a)(y) = w$ ,
- (iv) there exists a substitution  $\sigma : \Xi_R \rightarrow X$  such that:
  - $\sigma \mathbf{x}_i = x_i$  for  $i = 1, \dots, n$ , and
  - $\sigma[t] = t$ .

Then define:

$$\lambda_X(s)(c)(t) = \begin{cases} \beta \langle \delta_{i_j}(b_j)(\sigma \mathbf{y}_j) \rangle_{j=1..k} & \text{if } R \text{ fits } s, c, t \text{ with } \sigma, \\ 0 & \text{otherwise.} \end{cases}$$

We shall now prove that  $\lambda$  is natural in  $X$ . To this end, for any function  $g : X \rightarrow Z$ , any  $s = \mathbf{f}(x_1, \delta_1, \dots, x_n, \delta_n) \in \Sigma(X \times (\mathcal{F}_{\mathfrak{W}} X)^A)$ ,  $c \in A$  and  $t \in T_{\Sigma} Z$ , one must check that:

$$(\mathcal{F}_{\mathfrak{W}} T_{\Sigma} g)^A(\lambda_X(s))(c)(t) = \lambda_Y(\Sigma(g \times (\mathcal{F}_{\mathfrak{W}} g)^A)(s))(c)(t).$$

The left side of this equation is:

$$(*) = \sum_{\substack{r \in T_{\Sigma} X \\ \text{s.t. } g[r] = t}} \lambda_X(s)(c)(r)$$

and the right side:

$$(**) = \begin{cases} \beta \langle (\mathcal{F}_{\mathfrak{W}} g)^A(\delta_{i_j})(b_j)(\theta \mathbf{y}_j) \rangle_{j=1..k} & \text{if } R \text{ fits } g[s], c, t \text{ with } \theta : \Xi_R \rightarrow Z, \\ 0 & \text{otherwise.} \end{cases}$$

The expression in the first clause of  $(**)$  can be further rewritten as:

$$\begin{aligned} \beta \langle (\mathcal{F}_{\mathfrak{W}} g)^A(\delta_{i_j})(b_j)(\theta y_j) \rangle_{j=1..k} &= \beta \left\langle \sum_{y \in f(\theta y_j)} \delta_{i_j}(b_j)(y) \right\rangle_{j=1..k} = \\ &= \sum_{\substack{y_1, \dots, y_k \in X \\ \text{s.t. } gy_j = \theta y_j}} \beta \langle \delta_{i_j}(b_j)(y_j) \rangle_{j=1..k}; \end{aligned}$$

the second equality makes use of the multiadditivity of  $\beta$ .

Note now that if any of the conditions (i)-(iii) above fails, then  $R$  does not fit  $g[s], c, t$ , and it does not fit  $s, c, r$  for any  $r$  such that  $g[r] = t$ . If this is the case, both  $(*)$  and  $(**)$  equal 0 and the naturality equation holds, therefore it can be safely assumed that conditions (i)-(iii) hold. With this assumption,  $(**)$  can be rewritten as:

$$(**) = \begin{cases} \sum_{\substack{y_1, \dots, y_k \in X \\ \text{s.t. } gy_j = \theta y_j}} \beta \langle \delta_{i_j}(b_j)(y_j) \rangle_{j=1..k} & \text{if } \exists \theta : \Xi_R \rightarrow Z. \theta x_i = gx_i, \theta[t] = t \\ 0 & \text{otherwise.} \end{cases}$$

Recall that  $\theta$  above, if it exists, is unique. Now if  $\theta$  exists, then tuples  $y_1, \dots, y_k \in X$  such that  $gy_j = \theta y_j$  are in bijective correspondence with substitutions  $\sigma : \Xi_R \rightarrow X$  such that  $\sigma x_i = x_i$  and  $g[\sigma[t]] = t$ . Moreover, the existence of such  $\sigma$  implies that an appropriate  $\theta$  exists (take  $\theta = g \circ \sigma$ ). As a result, we can rewrite:

$$(**) = \sum_{\substack{\sigma : \Xi_R \rightarrow X \\ \text{s.t. } \sigma x_i = x_i, \\ g[\sigma[t]] = t}} \beta \langle \delta_{i_j}(b_j)(\sigma y_j) \rangle_{j=1..k}$$

Obviously, a substitution  $\sigma$  as above yields a term  $r \in T_\Sigma X$  such that  $g[r] = t$  (take  $r = \sigma[t]$ ). Moreover, for every  $r \in T_\Sigma X$  such that  $R$  fits  $s, c, r$  with a substitution  $\sigma$ , the substitution satisfies the condition in the sum above. As a result, now dropping the assumption that conditions (i)-(iii) hold, we may rewrite:

$$\begin{aligned} (**)&= \sum_{\substack{r \in T_\Sigma X \\ \text{s.t. } g[r] = t, \\ R \text{ fits } s, c, r}} \beta \langle \delta_{i_j}(b_j)(\sigma y_j) \rangle_{j=1..k} = \\ &= \sum_{\substack{r \in T_\Sigma X \\ \text{s.t. } g[r] = t}} \begin{cases} \beta \langle \delta_{i_j}(b_j)(\sigma y_j) \rangle_{j=1..k} & \text{if } R \text{ fits } s, c, r \text{ with } \sigma \\ 0 & \text{otherwise} \end{cases} = \\ &= \sum_{\substack{r \in T_\Sigma X \\ \text{s.t. } g[r] = t}} \lambda_X(s)(c)(r) = (*). \end{aligned}$$

This shows that a single  $\mathfrak{W}$ -GSOS rule defines an appropriate natural transformation. For an arbitrary  $\mathfrak{W}$ -GSOS specification  $\lambda$ , define

$$\lambda_X(s)(c)(t) = \sum_{R \in A} \lambda_X^R(s)(c)(t)$$

where  $\lambda^R$  arises from every single rule  $R$  as described above. Thanks to the finiteness condition in Definition 5, for each  $s$  and  $c$  the sum contains only finitely many non-zero summands, therefore the sum is well-defined and  $\lambda_X(s)(c)$  is finitely supported. Naturality of  $\lambda$  follows easily.

It remains to be seen that the  $\mathfrak{W}$ -LTS induced from a  $\mathfrak{W}$ -GSOS specification coincides with the  $(\mathcal{F}_{\mathfrak{W}} -)^A$ -coalgebra arising from the corresponding  $\lambda$  according to (2). To this end, it is enough to show that the induced LTS, seen as a coalgebra, makes (2) commute. A straightforward way to prove this is to notice that the induction step in Definition 6 corresponds exactly to the definition of  $\lambda$ , along the correspondence between  $\mathfrak{W}$ -LTSSs and  $(\mathcal{F}_{\mathfrak{W}})^A$ -coalgebras.  $\square$

# On the Specification and Verification of Model Transformations

Fernando Orejas<sup>1</sup> and Martin Wirsing<sup>2</sup>

<sup>1</sup> Universitat Politècnica de Catalunya, Barcelona, Spain  
orejas@lsi.upc.edu

<sup>2</sup> Ludwig-Maximilians Universität, Munich, Germany  
wirsing@pst.ifi.lmu.de

**Abstract.** Model transformation is one of the key notions in the model-driven engineering approach to software development. Most work in this area concentrates on designing methods and tools for defining or implementing transformations, on defining interesting specific classes of transformations, or on proving properties about given transformations, like confluence or termination. However little attention has been paid to the verification of transformations. In this sense, the aim of this work is, on one hand, to clarify what means to verify a model transformation and, on the other, to propose a specific approach for proving the correctness of transformations. More precisely, we use some general patterns to describe both the transformation and the properties that we may want to verify. Then, we provide a method for proving the correctness of a given transformation.

## 1 Introduction

Model transformation plays a central role in the model-driven engineering approach to software development. In this context, a model is an abstract description of an artifact, and model transformations are used to build new models out of existing ones. There are many kinds of model transformations that are considered interesting [17]. For instance, transformations going from more abstract to more concrete models may be part of the refinement process for yielding the final implementation of a system. Conversely, a transformation yielding more abstract models out of more concrete models may be used when doing reverse engineering, for example in connection with legacy systems. There are also transformations where the source and target models share the same level of abstraction. For example, this is the case of refactorings, where we change the structure of a system perhaps for improving its performance. From a different point of view, model transformations may be endogenous or exogenous. In endogenous transformations the source and the target model belong to the same class of models. This means that the transformation is defined within the same metamodel. This is the case, for example, of refactorings. Conversely, in an exogenous transformation the source and the target models may be of a very different kind. Exogenous transformations may occur for instance when the source and the target model describe a given system at different levels of abstraction.

Most work in the area of model transformation concentrates on designing methods and tools for defining or implementing transformations, on defining interesting specific

classes of transformations, or on proving properties about given transformations, like confluence or termination. However little attention has been paid to the problem of ensuring that a given transformation is correct. Where *correct* means that the source and the target models are in some sense a description of the same artifact. Possibly, one of the reasons why there is not much work on this topic is the difficulty, in the case of exogenous transformations, of having to deal with different kinds of models. Actually, one of the few approaches [26] that we know on the verification of model transformations concentrates on the case of endogenous transformations and, in particular, in the case of refactorings, although their techniques would be applicable to other kinds of endogenous transformations. The difficulty of dealing, on a uniform setting, with the verification of exogenous model transformations is covered in [2] by working at the level of *institutions* [10]. More precisely, in that paper the authors consider that in an exogenous transformation the source and the target models live on different institutions. Then, the correctness of a transformation is stated in terms of the existence of an intermediate institution that relates the source and target institutions through a span of an institution morphism and an institution comorphism. The idea is that the models in this intermediate institution include the common aspects of the source and the target models. The approach, which in some sense is conceptually clarifying and has influenced the present work, has two main problems from our point of view. On one hand, working at this multi-institutional level makes difficult to develop specific methods and tools for the verification of transformations. A solution to this problem, which is in a way implicit in [2] and which has been pursued further in [1], consists in the representation of all metamodels in a unifying framework, in this case Rewriting Logic. Unfortunately, as said above, this possible solution is only implicit in [2], while the work presented in [1] is, in a way, not directly related. Although in [1] they approach the verification problem, it is a different verification problem. More precisely, they are not concerned on whether a given transformation is correct in the above sense, but on how, by using their approach in connection with the Maude system, one can verify the resulting model, in the sense, for example, of showing that it is deadlock-free. The second problem of [2] is that they assume that it is possible to verify a model transformation by itself, without any additional information, by showing that the source and the target models are, in some sense, semantically equivalent. We believe that this is very reasonable for certain kinds of transformations, for example refactorings, but not in general. From our point of view, the correctness of a model transformation depends on what we are interested to observe in the source and target models. For instance, suppose that we have a model describing a system that processes some input coming from some components. Moreover, suppose that this model describes that this input is processed according to the order of arrival. Now, suppose that we apply a transformation to this model and, in the resulting target model, the input is not processed according to the order of arrival. This transformation would be incorrect if the order in which the input is processed is important, but it may be correct otherwise.

The work that we present is also influenced by the previous work of the first author on the use of graph constraints for modelling of software systems [22,21] and, especially, for the specification of model transformation [23,12]. More precisely, in [6], de Lara and Guerra introduced a visual declarative method for specifying bidirectional

model transformation based on the use of what they called *triple patterns*. This method was itself based on the use of triple graphs [27] to describe model transformations and on the use of graph constraints for software modelling. In [23] we showed how a specification specification of this kind could be translated into a finitely terminating graph transformation system, where the valid transformations were terminal graphs of the transformation system, and in [12] we extended the specification approach to deal with attributes over predefined data types, using a similar technique to [21]. In the current paper, we follow a similar approach to [12] for the specification of model transformations, but instead of considering that models are characterized by graphs, we generalize the approach dealing instead with algebras. The main reason is the limited expressive power of graph constraints and triple patterns. For example, if class diagrams are represented as typed graphs and the subclass relationship is represented by edges, then using graph constraints or triple patterns it is impossible to express that a certain class is a non-direct subclass of another class. This is not a problem when graphs are replaced by algebras. On the other hand, dealing with models as algebras would allow us to use a tool as MAUDE in the implementation of our results.

Nevertheless, the main contribution of this paper is related to the verification of model transformations, rather than with their specification. More precisely, after some conceptual discussion, we present a specific method, based also on the use of patterns, to specify the properties that characterize the correctness of a given transformation, and we develop the basis for a method for checking that a transformation is correct. To describe and motivate our concepts and results we use the standard example [25] of the transformation of class diagrams into relational schemas.

The paper is structured as follows. In the following section we present some basic algebraic notation and terminology at the same time that we discuss its use in the context of the model-driven engineering framework. Then, in Section 3 we introduce the notion of algebraic constraint as the translation of the notion of graph constraint when dealing with algebras instead of graphs. In this section we also introduce the category of triple algebras, following the ideas of the triple graph approach, and we show how we can use (triple) patterns to describe properties of a model transformation. Next, in Section 4 we generalize the specification method presented in [6,12] to the algebraic case. In the 5th section we address the problem of verifying model transformations. We discuss what it means to verify a model transformation and we propose a specific approach including, as mentioned above, a method for checking that a transformation is correct. Finally, in Section 6, we draw some conclusions and discuss further work.

## 2 Metamodels, Models and Instances

In the context of the Model-Driven Engineering approach (MDE), a model is a description of a system. This means that a model is what in different contexts is called a specification. As pointed out in [2] this may cause some confusion in terminology. Then, following [2], models in the sense of the MDE will often be called *software engineering models*. A software engineering model may describe the structure of a system. This is the case, for instance, of a class diagram. These models are called *structural models*. Otherwise, they may describe the behaviour of a system, like Petri nets or state

diagrams. In that case they are called *behavioral models*. In this paper we will essentially concentrate on structural models.

A class of software engineering models is defined by means of a *metamodel*. In MDE a metamodel essentially describes the structure of models, but not its semantics, which is left implicit. This is typically done by means of a class diagram. But this is not necessarily the only kind of metamodel specification. In [3,1] metamodels are defined by means of algebraic specifications using Maude [5]. In that approach, a signature plus some axioms play the same role as class diagrams to describe the structure of a model. Actually, they provide in this way a specification of class diagrams. In that context, the initial algebra of a (metamodel) specification can be seen as the class of all the software engineering models associated to the given metamodel. This means that every ground term over the given signature may be considered to denote a software engineering model. Metamodel definition is not relevant for the work presented in this paper. Therefore, we will not assume any concrete form of metamodel specification.

As said above, a software engineering model is a specification of a system. In the case of structural models, this specification describes the different kinds of parts of the given system. For instance, in a class diagram we specify the classes of objects that our system may include, and the relations between these classes. This means that, again, from an algebraic point of view, we can see a structural model as a signature, perhaps together with some axioms (for instance, an axiom may state that inheritance is transitive). In many cases, the signatures associated to a given kind of model may be just standard algebraic signatures including just some sorts, operations and, perhaps, relations. However, in some other cases, we may consider that the signatures associated to a given kind of model may be in some sense different, for instance they may need to have a richer type structure, like order-sorted signatures [11]. Anyhow, to be concrete and to provide a better intuition, we will assume that the signatures associated to the models considered are standard algebraic signatures. We think that this is not a real limitation. On one hand, using standard signatures, but having a semantics which is not the standard one, we may, in a way, code other kinds of models. For instance, in the approaches presented in [4,18,19,20] standard algebras are used, in some sense, to code algebras with a richer type structure. On the other hand, all our constructions and results are based on using some general categorical constructions and properties. This means that, if we replace the specific categories used in this paper by other categories which enjoy the same kind of constructions and properties, then the same results would still apply.

Therefore, we will consider that software engineering models consist of a *signature*  $\Sigma = (S, \Omega, \Pi)$  and, perhaps some  $\Sigma$ -axioms  $Ax$ , where  $S$  is the set of sorts of  $\Sigma$ ,  $Sorts(\Sigma)$ ,  $\Omega$  is the family of operators of  $\Sigma$ ,  $Ops(\Sigma)$ , and  $\Pi$  is the family of relation symbols of  $\Sigma$ ,  $Rel(\Sigma)$ . Then, a *signature morphism*  $f: \Sigma_1 \rightarrow \Sigma_2$  consists of three mappings,  $f_{Sorts}: Sorts(\Sigma_1) \rightarrow Sorts(\Sigma_2)$ ,  $f_{Ops}: Ops(\Sigma_1) \rightarrow Ops(\Sigma_2)$ , and  $f_{Rel}: Rel(\Sigma_1) \rightarrow Rel(\Sigma_2)$ , such that if the arity of  $\sigma_1 \in Ops(\Sigma_1)$  is  $s_1 \times \dots \times s_n \rightarrow s$  (resp. if the arity of  $\rho_1 \in Rel(\Sigma_1)$  is  $s_1 \times \dots \times s_n$ ) then the arity of  $f(\sigma)$  is  $f_{Sorts}(s_1) \times \dots \times f_{Sorts}(s_n) \rightarrow f_{Sorts}(s)$  (resp. the arity of  $f(\rho)$  is  $f_{Sorts}(s_1) \times \dots \times f_{Sorts}(s_n)$ ). Signatures and signature morphisms form the category **Sig**.

For instance, a class diagram may be seen as a signature, similar to a graph signature, having one sort for classes, one sort for associations and one sort for the inheritance relations together with four relation symbols  $SourceAssoc : Associations \times Classes$ ,  $TargetAssoc : Associations \times Classes$ ,  $SourceInh : Inheritance \times Classes$  and  $TargetInh : Inheritance \times Classes$ . In addition, we may consider an axiom stating that inheritance is transitive. An alternative specification would consist in representing class diagrams by signatures, where we have a sort for each class in the diagram, where associations and inheritance relations are represented as relation symbols in the signature, and where attributes and methods are represented as operators. This is essentially how class diagrams are represented in [3,1]. Nevertheless, in this paper we will not define any preferred signature for class diagrams. Instead, in the examples we will directly present them in the standard graphical representation.

The possible states of a system defined by a given software engineering model  $(\Sigma, Ax)$  are often called the *instances* of the model. Using the standard terminology from mathematical logic, instances would be models of  $(\Sigma, Ax)$ , i.e.  $\Sigma$ -algebras that satisfy the axioms in  $Ax$ .

To be more precise, a  $\Sigma$ -algebra  $A$ , as usual, consists of an S-indexed family of carriers,  $\{A_s\}_{s \in S}$ , an interpretation  $\sigma_A : A_{s_1} \times \dots \times A_{s_n} \rightarrow A_s$  for each operator  $\sigma : s_1 \times \dots \times s_n \rightarrow s$  in  $\Omega$ , and an interpretation  $p_A : A_{s_1} \times \dots \times A_{s_n} \rightarrow A_s$  for each relation symbol  $p : s_1 \times \dots \times s_n \rightarrow s$  in  $\Pi$ . A  $\Sigma$ -homomorphism  $h : A \rightarrow B$  is an S-indexed family of mappings,  $\{f_s : A_s \rightarrow B_s\}_{s \in S}$  commuting with the operations and relations in  $\Sigma$ .  $\Sigma$ -algebras, together with the  $\Sigma$ -homomorphisms form the category,  $\mathbf{Alg}_\Sigma$ .

In this paper we will have to relate algebras with different signatures. This is done through the notions of *forgetful functor* and *generalized algebra morphism*. In particular, as it is well known, every signature morphism  $f : \Sigma_1 \rightarrow \Sigma_2$  has an associated forgetful functor  $U_f : \mathbf{Alg}_{\Sigma_2} \rightarrow \mathbf{Alg}_{\Sigma_1}$ . If  $f$  is a signature inclusion we may write  $A_2|_{\Sigma_1}$  instead of  $U_f(A_2)$ . Then, a generalized algebra morphism (or, just, an algebra morphism)  $h : A_1 \rightarrow A_2$ , where  $A_1 \in \mathbf{Alg}_{\Sigma_1}$  and  $A_2 \in \mathbf{Alg}_{\Sigma_2}$  consists of a signature morphism  $h_{Sig} : \Sigma_1 \rightarrow \Sigma_2$  and of a  $\Sigma_1$ -homomorphism  $h_{Alg} : A_1 \rightarrow U_{h_{Sig}}(A_2)$ . A generalized morphism  $h$  where both  $h_{Sig}$  and  $h_{Alg}$  are inclusions will be called an algebra embedding, or just an embedding. The class of all algebras, together with the generalized morphisms form the category,  $\mathbf{Alg}$ . In [7] it is proved that  $\mathbf{Alg}$  has pushouts. We will not consider any specific logic for writing the axioms associated to a model specification. It may be first-order logic, or some fragment of first-order logic, or a different kind of logic. We will only assume that if  $\text{Form}_\Sigma$  is the set of all  $\Sigma$ -formulas then:

- There is a well-defined satisfaction relation between  $\Sigma$ -algebras and  $\Sigma$ -formulas, for each given assignment of the free variables involved.
- Signature morphisms induce formula translations, i.e. for every morphism  $f : \Sigma_1 \rightarrow \Sigma_2$  there is an associated mapping  $f^* : \text{Form}_{\Sigma_1} \rightarrow \text{Form}_{\Sigma_2}$ .
- The so-called satisfaction property of institutions holds. This means that for each signature morphism  $f : \Sigma_1 \rightarrow \Sigma_2$ , each sentence  $\alpha \in \text{Form}_{\Sigma_1}$  and each algebra  $A \in \mathbf{Alg}_{\Sigma_2}$  we have that  $A \models f^*(\alpha)$  if and only if  $U_f(A) \models \alpha$ .

In these conditions, we may extend the previous constructions to deal with specifications (or software engineering models). In particular, given  $SP = (\Sigma, Ax)$ , we will denote

by  $\mathbf{Alg}_{SP}$  the full subcategory of  $\mathbf{Alg}_\Sigma$  consisting of all algebras satisfying the formulas in  $Ax$ . Also, we will consider that a specification morphism  $f: (\Sigma_1, Ax_1) \rightarrow (\Sigma_2, Ax_2)$  is just a signature morphism  $f: \Sigma_1 \rightarrow \Sigma_2$  such that for every  $\alpha_1 \in Ax_1$ , we have that  $f^*(\alpha_1) \in Ax_2$ . Specifications and specification morphisms form the category of specifications  $\mathbf{Spec}$ . Moreover, every specification morphism  $f: (\Sigma_1, Ax_1) \rightarrow (\Sigma_2, Ax_2)$  has an associated forgetful functor  $U_f: \mathbf{Alg}_{(\Sigma_2, Ax_2)} \rightarrow \mathbf{Alg}_{(\Sigma_1, Ax_1)}$ .

As said above, any software engineering model  $SP = (\Sigma, Ax)$  is assumed to denote a class of instances, which are  $\Sigma$ -algebras that satisfy the axioms in  $Ax$ . However, not necessarily every algebra in  $\mathbf{Alg}_{SP}$  needs to be considered a valid instance of  $SP$ . This means that we will consider that every software engineering model  $SP = (\Sigma, Ax)$  denotes a category of algebras  $\mathbf{Inst}_{SP}$ , which is a subcategory of  $\mathbf{Alg}_{SP}$ . For example, from now on we will consider that every instance of a software engineering model extends a basic data algebra  $D$ . This means that given a model  $SP = (\Sigma, Ax)$  we assume that  $\Sigma_{data} \subseteq \Sigma$ , where  $\Sigma_{data}$  is a given data signature. We also assume given a data algebra  $D \in \mathbf{Alg}_{\Sigma_{data}}$  such that for each  $A \in \mathbf{Inst}_{SP}$  we have  $A|_{\Sigma_{data}} = D$ . Moreover, if  $h: A \rightarrow B$  in a homomorphism in  $\mathbf{Inst}_{SP}$  then  $h$  is data-preserving, which means that  $h|_{\Sigma_{data}}: A|_{\Sigma_{data}} \rightarrow B|_{\Sigma_{data}}$  is the identity. In the same sense, we say that a signature morphism  $f: \Sigma_1 \rightarrow \Sigma_2$  is data-preserving if both signatures  $\Sigma_1$  and  $\Sigma_2$  include  $\Sigma_{data}$  and  $f$  is the equality when restricted to  $\Sigma_{data}$ . Similarly, a generalized algebra morphism  $h: A_1 \rightarrow A_2$  is data-preserving if  $h_{Sig}$  is data-preserving and  $h|_{\Sigma_{data}}: A_1|_{\Sigma_{data}} \rightarrow U_{h_{Sig}}(A_2)|_{\Sigma_{data}}$  is the identity.

A property that we use to prove our results is pair factorization (actually, its general version,  $n$ -factorization). More precisely,  $n$  morphisms,  $f_i: a_i \rightarrow c$ , for  $1 \leq i \leq n$ , are *jointly epimorphic* if for all morphisms  $f, g: c \rightarrow d$ , we have that  $f \circ h_i = g \circ h_i$ , for every  $i$ , implies  $f = g$ . A category has the  *$n$ -factorization property* if for every family of morphisms  $\{a_1 \xrightarrow{f_1} a, \dots, a_n \xrightarrow{f_n} a\}$ , with the same codomain  $a$  there exists an object  $b$ , a monomorphism  $m$  and a jointly surjective family of morphisms  $\{a_1 \xrightarrow{g_1} b, \dots, a_n \xrightarrow{g_n} b\}$  such that the diagram below commutes:

$$\begin{array}{ccccc}
 & a_1 & & & \\
 & \searrow f_1 & & & \\
 & g_1 & \nearrow & & \\
 & \vdots & & & \\
 & g_n & \nearrow & & \\
 & a_n & & & \\
 & & \nearrow f_n & & \\
 & & & b \xrightarrow{m} a & \\
 & & & \swarrow & \\
 & & & & 
 \end{array}$$

It must be noted that if  $\{f_1, \dots, f_n\}$  are monomorphisms then so are  $\{g_1, \dots, g_n\}$ . As a consequence, it is easy to prove the following proposition:

**Proposition 1. ( $n$ -factorization)** *The categories  $\mathbf{Sig}$ ,  $\mathbf{Spec}$ ,  $\mathbf{Alg}_\Sigma$ ,  $\mathbf{Alg}_{SP}$ , and  $\mathbf{Alg}$  have the  $n$ -factorization property, moreover if the morphisms  $f_i$  are data-preserving so are the  $g_i$  and  $m$ .*

*Proof.* Given the signature morphisms  $f_i: \Sigma_i \rightarrow \Sigma$ , we define  $\Sigma'$  as the subsignature of  $\Sigma$  that includes all the sorts, operators and relation symbols in the images of the  $f_i$ . Then, we have that the diagram below commutes:

$$\begin{array}{ccccc}
 & & \Sigma_1 & & \\
 & \swarrow & f_1 & \searrow & \\
 & g_1 & & & \\
 & \vdots & & & \\
 & \swarrow & g_n & \searrow & \\
 & \Sigma_n & & & \\
 & \nearrow & f_n & & \\
 & & \Sigma' & \xrightarrow{m} & \Sigma
 \end{array}$$

where  $m$  is the inclusion morphism. The proofs for  $\mathbf{Spec}$ ,  $\mathbf{Alg}_\Sigma$ ,  $\mathbf{Alg}_{SP}$ , and  $\mathbf{Alg}$  are similar. ■

In addition, we will assume that for any software engineering model  $SP$ ,  $\mathbf{Inst}_{SP}$  also has the  $n$ -factorization property.

### 3 Algebraic Patterns and Triple Algebras

In this section we introduce the two basic constructions that are used to specify model transformations. First, we introduce the notion of algebraic patterns, as a generalization of the notion of graph constraints or graph conditions [9,13,15,14,21]. Then we present the notion of triple algebras, which is the corresponding generalization of the notion of triple graphs, introduced by Schürr [27].

#### 3.1 Algebraic Patterns

Graph constraints were introduced in the area of graph transformation, together with the notion of (negative) application conditions, as a way to limit the applicability of transformation rules [9,13,15,14]. The underlying idea of a graph constraint is that it should specify that certain patterns must be present (or must not be present) in a given graph. However, later, several authors [16,22,24] have shown how graph constraints may be used for visual modelling. The simplest kind of constraint, which we called basic constraints in [22], is just a graph  $C$ . If  $C$  is considered to be a positive constraint then a graph  $G$  would satisfy  $C$  if it includes a copy of  $C$ . Conversely, if  $C$  is considered to be a negative constraint (or the negation of a positive constraint) then a graph  $G$  would satisfy  $C$  if it does not include any copy of  $C$ . For example, given the constraint  $C$ :



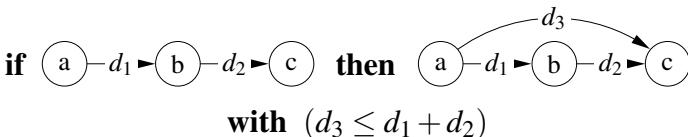
if the constraint is considered to be positive then it would specify that a given graph  $G$  should include at least two nodes with two edges connecting them. Conversely, if  $C$  is considered to be a negative constraint then it would specify that a given graph  $G$  should

not include two different edges between any two nodes. A slightly more complex kind of graph constraints are graph monomorphisms (or, just, inclusions)  $c : X \rightarrow C$ , which we represent pictorially using an **if - then** notation, where the two graphs involved have been labelled to implicitly represent the given monomorphism. These constraints are sometimes called conditional constraints and are a generalization of basic constraints: the constraint  $C$  is equivalent to the constraint  $c : X \rightarrow C$  when  $X$  is the empty graph. If the constraint  $c$  is considered to be positive then it specifies that whenever a graph  $G$  includes (a copy of) the graph  $X$  it should also include (a copy of) its extension  $C$ . If  $c$  is considered to be a negative constraint then it would specify that the graph  $G$  includes a copy of the graph  $X$  but not a copy of its extension  $C$ . For instance, if the constraint  $c$ :



is a positive constraint then it would specify that a graph must be transitive, i.e. the constraint says that for every three nodes,  $a, b, c$  if there is an edge from  $a$  to  $b$  and an edge from  $b$  to  $c$  then there should be an edge from  $a$  to  $c$ . If  $c$  is negative then it would specify that a graph must not be transitive. In what follows, satisfaction of negative constraints is just the negation of satisfaction of positive constraints, and to avoid confusion between positive and negative constraint, we will write negative constraints using the negation symbol. This means that a constraint  $C$  will be assumed to be positive, while a constraint  $\neg C$  will be assumed to be a negative constraint.

If we want to define constraints on attributed graphs, i.e. graphs that can *store* values from a given data algebra, in principle, we can use the same kind of constructions. This means that a graph constraint would be either a graph  $C$  or an inclusion  $c : X \rightarrow C$ , where  $X$  and  $C$  are attributed graphs. However, as discussed in [21], this poses some problems. On one hand, it may be difficult (if not impossible) to express constraints that include (or depend on) conditions on the data values. On the other, the formal definition of attributed graphs [8] is a bit complex. This means that further technical development becomes complicated. To avoid this problems, in [21], a solution inspired in the area of Constraint Logic Programming is proposed. The idea is to separate the *graph part* in the constraints from the *data part*. This is done by considering that a graph constraint consists of a graph labelled with some variables (respectively, a morphism between labelled graphs) and formulas expressing a condition on these variables. For instance, the constraint below:



may express that, in a graph representing the distances between some cities, these distances must satisfy the so-called triangular property.

It should be mentioned that this separation of concerns, between the graph and the data part of a graph constraint, may also facilitate the construction of deductive and verification tools. In particular it would allow us to reuse existing powerful and efficient constraint solvers to deal with the data part of constraints.

Algebraic patterns, as presented in this paper, are inspired by, and generalize, attributed graph constraints. The basic idea is similar: an algebraic pattern consists of an algebra with variables (instead of data) (or an embedding between algebras with variables) and a formula expressing some conditions on these variables. This is defined as follows: given a (data-sorted) set of variables  $X$ , first, we define a signature of variables  $\hat{X}$  whose only operation symbols are the variables (seen as constants); next, for each signature  $\Sigma$  extending  $\Sigma_{data}$  we define the signature  $\Sigma(X)$ , obtained replacing in  $\Sigma$  the operators and relation symbols in  $\Sigma_{data}$  by the variables in  $X$ ; then, we define an algebra of variables  $\aleph_X$  whose only elements are the variables in  $X$ ; finally, a pattern will just be an algebra extending  $\aleph_X$  or an embedding over algebras extending  $\aleph_X$ , together with some  $\Sigma_{data}$ -formula on the variables. From now on we will assume that the given sets of variables are finite.

**Definition 1 (Signature of variables, Algebra of variables).** *Given a data signature  $\Sigma_{data}$  and an  $S_{data}$ -sorted finite set of variables  $X$ , we define the signature of variables over  $X$ ,  $\hat{X}$ , as the signature  $(S_{data}, X, \emptyset)$ . Given a signature  $\Sigma$  extending  $\Sigma_{data}$ , the extension of  $\Sigma$  over  $X$ ,  $\Sigma(X)$  is defined by the pushout in the diagram below:*

$$\begin{array}{ccc} (S_{data}, \emptyset, \emptyset) & \longrightarrow & \hat{X} \\ \downarrow & \text{po} & \downarrow \\ \Sigma \setminus \Sigma_{data} & \longrightarrow & \Sigma(X) \end{array}$$

where  $\Sigma \setminus \Sigma_{data}$  denotes the signature where all the operators and relation symbols (but not the sorts) in  $\Sigma_{data}$  have been removed from  $\Sigma$ .

We also define the algebra of variables over  $X$ ,  $\aleph_X$ , as the term algebra over  $\hat{X}$ ,  $T_{\hat{X}}$ .

Notice that every generalized morphism from the algebra of variables  $\aleph_X$  into the data algebra  $D$  can be seen as a variable assignment, and vice versa.

**Definition 2 ( $\Sigma(X)$ -algebras with variables, Algebraic patterns).** *Given an  $S_{data}$ -sorted set of variables  $X$ , and a signature  $\Sigma$  extending  $\Sigma_{data}$ , a  $\Sigma(X)$ -algebra with variables in  $X$  is a finite  $\Sigma(X)$ -algebra extending  $\aleph_X$ . Given a  $\Sigma_1(X)$ -algebra  $A_1$  and a  $\Sigma_2(X)$ -algebra  $A_2$ , a generalized morphism with variables  $h : A_1 \rightarrow A_2$  is a generalized morphism such that  $h_{Alg}|_{\hat{X}}$  is the equality. As before, generalized morphisms with variables whose signature and algebra parts are inclusions are called embeddings.*

A basic algebraic pattern  $P$ , is a 3-tuple  $(X, C, \alpha)$ , where  $X$  is a data-sorted set of variables,  $C$  is a  $\Sigma(X)$ -algebra with variables and  $\alpha$  is a  $\Sigma_{data}$ -formula with free variables in  $X$ .

A conditional algebraic pattern  $CP$  is a 5-tuple  $(X_1, X_2, c, \alpha_1, \alpha_2)$ , where  $X_1$  and  $X_2$  are data-sorted sets of variables,  $c : C_1 \rightarrow C_2$  is an embedding with variables,  $C_1$  and  $C_2$  are  $\Sigma_1(X_1)$  and  $\Sigma_2(X_2)$ -algebras, respectively, and  $\alpha_1$  and  $\alpha_2$  are  $\Sigma_{data}$ -formulas with free variables in  $X_1$  and  $X_2$ , respectively.

For instance, let us consider that a graph (an attributed graph) is an algebra over a signature with two sorts, *Nodes* and *Edges* (in addition to the data sorts), and two operations *source*: *Edges* → *Nodes* and *target*: *Edges* → *Nodes* (in addition to the operations in  $\Sigma_{data}$ ). In that case, the above examples of graph constraints could be considered examples of algebraic patterns.

Satisfaction of algebraic patterns is just the straightforward generalization of graph constraint satisfaction. In particular, Given a  $\Sigma(X)$ -algebra  $A$  (positively) satisfies the basic pattern  $(C, \alpha)$  if there is a generalized monomorphism  $h : C \rightarrow A$  such that the data algebra  $D$  satisfies the formula  $\alpha$  when the variables in  $X$  have been replaced by their images in  $h$ . Satisfaction of conditional patterns is the corresponding generalization. Satisfaction of negative patterns is just the negation of positive satisfaction.

**Definition 3 (Satisfaction of algebraic patterns).** A  $\Sigma$ -algebra  $A$  satisfies the basic pattern  $(X, C, \alpha)$ , denoted  $A \models (X, C, \alpha)$  if there is a generalized monomorphism  $h : C \rightarrow A$  such that  $D \models_{h_{Alg}|_X} \alpha$ .

A  $\Sigma$ -algebra  $A$  satisfies the conditional algebraic pattern  $(X_1, X_2, c, \alpha_1, \alpha_2)$  iff for each generalized monomorphism  $m : C_1 \rightarrow A$  such that  $D \models_{m_{Alg}|_{\widehat{X}_1}} \alpha_1$  there is a generalized monomorphism  $m' : C_2 \rightarrow A$  such that  $m = m' \circ c$  and  $D \models_{m'_{Alg}|_{\widehat{X}_2}} \alpha_2$ .

### 3.2 Triple Algebras and Triple Patterns

Triple graphs were defined by Schürr [27] to describe model transformations when models and instances are represented as graphs. Instead of considering that a model transformation is just characterized by the source and target model, Schürr considered that it is very important to establish a connection between the corresponding elements of the source and target graph. The way to do this is by using an intermediate graph, the *connection graph*, and one mapping from this connection graph into each of the source and target graphs. Here, we use the same idea, but using algebras instead of graphs.

**Definition 4 (Triple algebras and Morphisms).** A triple algebra  $TrA = (TrA_S \xleftarrow{c_S} TrA_C \xrightarrow{c_T} TrA_T)$  (or just  $TrA = \langle TrA_S, TrA_C, TrA_T \rangle$  if  $c_S$  and  $c_T$  may be considered implicit) consists of three algebras  $TrA_S$ ,  $TrA_C$ , and  $TrA_T$ , called the source, connection and target algebras, respectively, and two generalized morphisms  $c_S$  and  $c_T$ . We say that  $TrA$  extends the data algebra  $D$  if the three algebras  $TrA_S, TrA_C, TrA_T$  extend  $D$  and the  $\Sigma_{data}$  reducts of the morphisms  $c_S$  and  $c_T$  coincide with the identity. A triple algebra morphism  $m = (m_S, m_C, m_T) : TrA^1 \rightarrow TrA^2$  consists of three generalized morphisms  $m_S$ ,  $m_C$ , and  $m_T$  such that  $m_S \circ c_S^1 = c_S^2 \circ m_C$  and  $m_T \circ c_T^1 = c_T^2 \circ m_C$ . In addition,  $m$  is data-preserving if the  $\Sigma_{data}$  reducts of the morphisms  $m_S$ ,  $m_C$  and  $m_T$  coincide with the identity.

Given a triple algebra  $TrA$ , we write  $TrA|_K$  for  $K \in \{S, C, T\}$  to refer to a triple algebra where only the  $K$  algebra is present and the other two components are the empty algebra over the empty signature, i.e.,  $TrA|_S = \langle TrA_S, \emptyset, \emptyset \rangle$ , and given a triple algebra morphism  $h : TrA_1 \rightarrow TrA_2$  we also write  $h|_K : TrA_1|_K \rightarrow TrA_2|_K$  to denote the morphism whose  $K$ -component coincides with  $h_K$  and whose other two components are the empty morphism between empty algebras. Finally, given  $TrA$ , we write  $i_G^K : TrA|_K \rightarrow TrA$  to denote the inclusion  $i_{TrA}^K : TrA|_K \rightarrow TrA$ , where the  $K$ -component is the identity.

Triple algebras form the category **TrAlg**, which can be constructed as the functor category  $\text{Alg}^{\leftarrow\rightarrow}$ .

In the same way that algebraic patterns describe properties that may be satisfied by an algebra, triple algebraic patterns describe properties that may be satisfied by a triple algebra. More precisely, first we may define the notions of triple algebra with variables and triple morphism with variables:

**Definition 5 (Triple algebras and Morphisms with variables).** A triple algebra with variables in  $X$ ,  $TrA = (TrA_S \xleftarrow{c_S} TrA_C \xrightarrow{c_T} TrA_T)$  is a triple algebra where  $TrA_S$ ,  $TrA_C$  and  $TrA_T$  are algebras with variables in  $X$  and  $c_S$  and  $c_T$  are generalized morphisms with variables.

A triple algebra morphism  $m = (m_S, m_C, m_T) : TrA^1 \rightarrow TrA^2$  is a triple morphism such that  $m_S, m_C$  and  $m_T$  are morphisms with variables.

It must be noted that if  $m = (m_S, m_C, m_T) : TrA^1 \rightarrow TrA^2$  is a data-preserving triple morphism, where  $TrA^1 = (TrA_S^1 \xleftarrow{c_S^1} TrA_C^1 \xrightarrow{c_T^1} TrA_T^1)$  is a triple algebra with variables in  $X$  and  $TrA^2 = (TrA_S^2 \xleftarrow{c_S^2} TrA_C^2 \xrightarrow{c_T^2} TrA_T^2)$  is a triple algebra that extends  $D$ , then  $m$  maps every variable in  $X$  to a unique data value. In particular, if  $x$  is a variable in  $X$  then  $x$  is a constant in the signatures of  $TrA_S^1$ ,  $TrA_C^1$ , and  $TrA_T^1$ . Now, let  $d_1 = m_S(x_{TrA_S^1})$ ,  $d_2 = m_C(x_{TrA_C^1})$ , and  $d_3 = m_T(x_{TrA_T^1})$ . Let us prove that  $d_1 = d_2$ :

$d_1 = m_S(x_{TrA_S^1}) = m_S(c_S^1(x_{TrA_C^1})) = c_S^2(m_C(x_{TrA_C^1})) = c_S^2(d_2)$ . But, since  $c_S^2$  is assumed to preserve data, this means that  $d_1 = d_2$ . The proof that  $d_2 = d_3$  is similar. Therefore, we may identify  $m$  with a variable assignment.

Now, we extend our notion of algebraic pattern to the notion of triple algebraic pattern, together with the corresponding definition of satisfaction:

**Definition 6 (Triple algebraic patterns).** A basic triple algebraic pattern  $P$ , is a 3-tuple  $(X, TrC, \alpha)$ , where  $X$  is a data-sorted set of variables,  $TrC$  is a triple algebra with variables in  $X$  and  $\alpha$  is a  $\Sigma_{\text{data}}$ -formula with free variables in  $X$ .

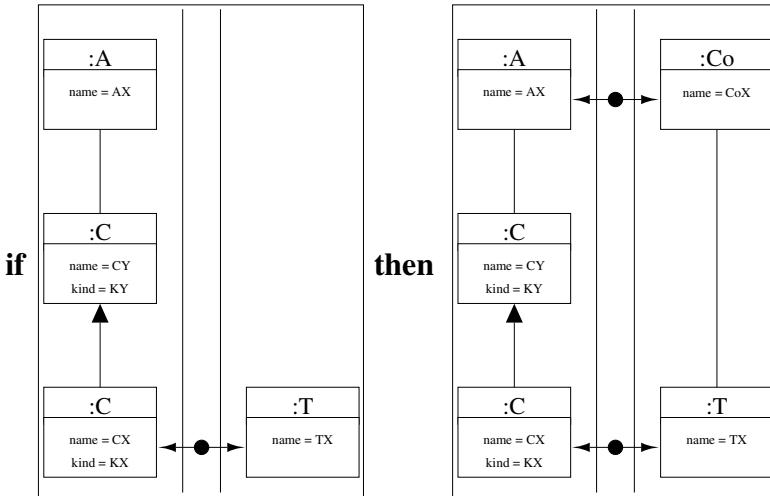
A conditional triple algebraic pattern  $CP$  is a 5-tuple  $(X_1, X_2, c, \alpha_1, \alpha_2)$ , where  $X_1$  and  $X_2$  are data-sorted sets of variables,  $c$  is a triple monomorphism with variables and  $\alpha_1$  and  $\alpha_2$  are  $\Sigma_{\text{data}}$ -formulas with free variables in  $X_1$  and  $X_2$ , respectively.

**Definition 7 (Satisfaction of triple algebraic patterns).** A triple algebra  $TrA$  that extends  $D$  satisfies the basic triple pattern  $(X, TrC, \alpha)$ , denoted  $A \models (X, TrC, \alpha)$  if there is a data-preserving triple monomorphism  $h : TrC \rightarrow TrA$  such that  $D \models_h \alpha$ .

A triple algebra  $TrA$  that extends  $D$  satisfies the conditional triple algebraic pattern  $(X_1, X_2, c, \alpha_1, \alpha_2)$ , if for each data-preserving triple monomorphism  $m : TrC_1 \rightarrow TrA$  such that  $D \models_m \alpha_1$  there is a data-preserving triple monomorphism  $m' : TrC_2 \rightarrow TrA$  such that  $m = m' \circ c$  and  $D \models_{m'} \alpha_2$ .

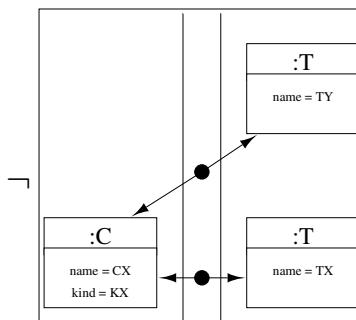
*Example 1.* A standard example in model transformation is the transformation of class diagrams into relational schemas [25]. According to this transformation (persistent) classes are transformed into tables and attributes are transformed into columns. Then, if a class  $c_1$  is a subclass of a class  $c_2$  and  $c_2$  has a certain attribute  $at$ , we know that

$c_1$  inherits that attribute. This means that the table associated to  $c_1$  should include a column associated to  $at$ . The pattern specifying this property, which this transformation must satisfy, is depicted below.



More precisely, this pattern specifies that whenever a triple algebra includes a class  $CX$ , which is a subclass of  $CY$ , and such that  $CY$  has an associated attribute  $X$  and  $CX$  is transformed into the table  $TX$ , then that triple algebra must also include a column  $CoX$ , which is associated to  $TX$  and is the transformation of  $AX$ . It may be noted that the above pattern does not include any formula on the variables  $CX, KX, KY, CY, AX, TX$  and  $CoX$ . The reason is that in this case there is no specific condition needed. Hence the formula is empty.

Another property that our transformation must satisfy is that a given class should not be transformed into two different tables. This may be specified by the negative pattern below:



In [22,21] it is shown how we can reason with graph constraints. In particular, sound and complete proof systems are presented which apply to constraints over a large class of graphical structures, including triple graphs. We believe that the results presented in these papers could also be applied to the case of algebraic patterns and triple algebraic patterns when the algebras involved in the patterns are finite.

## 4 Specification of Model Transformations by Triple Patterns

A model transformation is a procedure that given instances of a source model yields instances of a target model. This procedure may be described algorithmically or declaratively. In this work we are concerned with the declarative description of a model transformation. In our context, specifying a model transformation means describing a class of triple algebras  $\langle TrA_S, TrA_C, TrA_T \rangle$ , where  $TrA_S$  is an instance of the source model and  $TrA_T$  is an instance of the target model. In particular, we could use triple algebraic patterns, as described in the previous section, to write this kind of specifications. More precisely, this means considering that a model transformation specification would be a set of positive and negative patterns (or, more generally, a set of axioms consisting of logical formulas whose atomic components are triple algebraic patterns). Then, a specification would denote, as usual, the class of all triple algebras satisfying these axioms. In that context, techniques similar to those developed in [22,21] would allow us, not only to reason about our specifications, including checking their consistency, but also to build minimal triple algebras satisfying the specification. Which means that we could have a way of automatically implementing the model transformation. However, in our opinion, an approach like the one introduced in [6] and further developed in [23,12], also based on the use of triple patterns, may be more appropriate for this purpose. On one hand, that approach is in a way close to the declarative fragment of OMG's standard language QVT (QVT-relational) [25] and may be found easier and more intuitive to use for defining specific transformations. The reason is that, even if the approach is essentially declarative, in a sense it reflects how one would construct the transformation. On the other hand, and partly as a consequence of the previous reason, that approach seems more appropriate for its (automatic) implementation. In particular, in [23], we have shown that, in the case of triple graphs and triple graph patterns, a model transformation specification in the sense of [6] may be converted into a finitely terminating graph transformation system where all the models (in the logical sense) of the specification are terminal graphs of the transformation system.

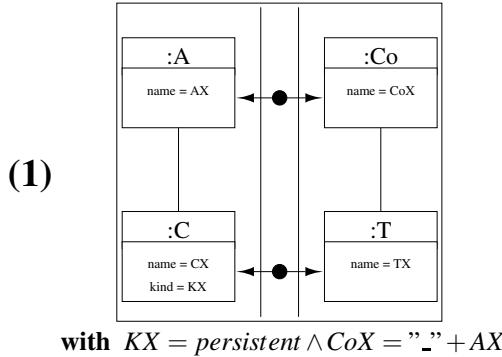
Patterns, as presented in [6,23,12], from now on called *transformation patterns*, have a similar syntax but different semantics than that of the patterns presented in the above section. The basic idea of transformation patterns is (in a way) to see them as tiles that have to “cover” a given source model, perhaps with some overlapping. The target model obtained by gluing the target parts of these patterns is the result of the transformation. In particular, this is formally stated by saying that a triple instance (forward)<sup>1</sup> satisfies

---

<sup>1</sup> Triple patterns may be considered to specify bidirectional transformations, i.e. source-to-target transformations and target-to-source transformations. In this sense, there is a corresponding notion of backward satisfaction. Actually, most of the associated notions and constructions that are presented below have a corresponding backward version.

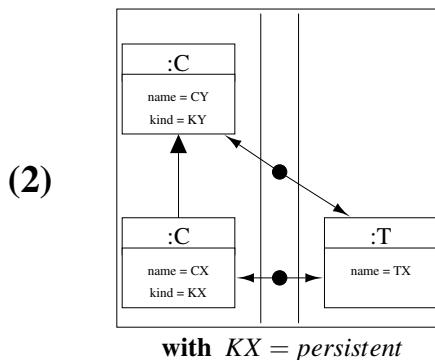
a transformation pattern if whenever the source part of the instance embeds the source part of the pattern then the whole instance should embed the pattern. Let us provide some intuition with an example:

*Example 2.* Let us suppose that we want to specify the transformation of class diagrams into relational schemas [25]. According to this transformation, persistent classes are transformed into tables and their associated attributes transformed into columns of these tables. This may be specified by the pattern below

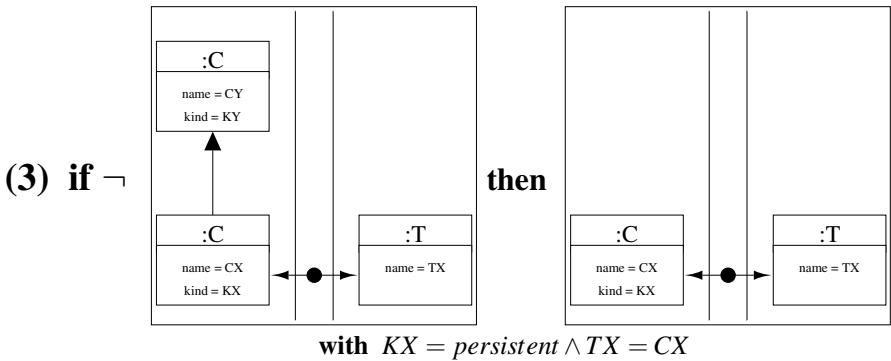


Then a triple instance would satisfy this pattern if, whenever there is a class in the source instance whose kind is persistent, and this class has an associated attribute, then the triple instance also includes a table and an associated column which are the transformation, respectively, of the class and the attribute. In this case, the formula in the pattern states that the kind of the class must be persistent and that the name of the column is obtained by adding an underline character in front of the name of the attribute.

Another pattern may specify that subclasses and parent classes are transformed into the same table:



Transformation patterns may also have a condition. In this case, the condition relates to the context of the pattern and is always negative. It plays a similar role to negative application conditions in graph transformation. Let us see an example:



The above pattern specifies that every persistent class if it does not have a parent class then it must be transformed into a table with the same name as the class. Notice that the condition includes the conclusion of the pattern. The reason is that the condition always refers to the context of the pattern.

In addition, negative basic patterns like the one presented in Example 1 may be used when specifying a transformation to describe relationships that should not occur between the source and target models.

**Definition 8 (Transformation patterns, transformation specifications).** A transformation pattern  $TP = \langle X, \{TrC \xrightarrow{n_j} N_j\}_{j \in J}, TrC, \alpha \rangle$  consists of

- A data-sorted set of variables  $X$ .
- The postcondition, given by the triple algebra  $TrC$  with variables in  $X$ .
- The negative preconditions, given by the embeddings with variables  $TrC \xrightarrow{n_j} N_j$ , for each  $j \in J$ .
- A data condition given by  $\alpha$ , which is a  $\Sigma_{\text{data}}$ -formula with free variables in  $X$ .

A transformation specification  $TSP$  is a finite set of transformation patterns and negative algebraic basic patterns.

Then, satisfaction is defined as described in the example. More precisely, a triple algebra  $TrA$  satisfies a transformation pattern  $TP = \langle X, \{TrC \xrightarrow{n_j} N_j\}_{j \in J}, TrC, \alpha \rangle$  if whenever there is a monomorphism  $m$  from  $TrC_S$  into the source of  $TrA$ , the preconditions are satisfied by  $m$  and the formula  $\alpha$  is satisfied by the data algebra with respect to the assignment defined by  $m$ , then there is a monomorphism from  $TrC$  into  $TrA$  extending  $m$ . And a monomorphism  $m$  of  $TrC_S$  in  $TrA_S$  satisfies a precondition  $TrC \xrightarrow{n_j} N_j$  if there is no embedding  $m'$  of  $(N_j)_S$  in  $TrA_S$  such that  $m'$  extends  $m$ . Then, we may consider that the semantics of a transformation specification  $TSP$  is the class of all triple algebras that satisfy the patterns in  $TSP$ .

**Definition 9 (Satisfaction of transformation patterns).**

A monomorphism  $m : TrC|_S \rightarrow TrA$  satisfies a negative precondition  $TrC \xrightarrow{n_j} N_j$ , denoted  $m \models TrC \xrightarrow{n_j} N_j$  if there does not exist a monomorphism  $g : (N_j)|_S \rightarrow TrA$  such that  $m = g \circ (n_j)|_S$ .

$TrA$  satisfies a transformation pattern  $TP = \langle X, \{TrC \xrightarrow{n_j} N_j\}_{j \in J}, TrC, \alpha \rangle$ , denoted  $TrA \models TP$ , if for every monomorphism  $m : TrC|_S \rightarrow TrA$ , such that for every  $j$  in  $J$   $m \models TrC \xrightarrow{n_j} N_j$  and such that  $D \models_m \alpha$ , there exist a monomorphism  $m' : TrC \rightarrow TrA$  such that  $m = m' \circ i_{TrC^S}$ :

$$\begin{array}{ccccc} & & (N_j)|_S & & \\ & \xleftarrow{(n_j)|_S} & TrC|_S & \xrightarrow{i_{TrC^S}} & TrC \\ & \searrow g / & m \downarrow & & \swarrow m' \\ & & TrA & & \end{array}$$

A triple graph  $TrA$  satisfies a negative basic pattern  $\neg(X, TrC, \alpha)$ , denoted  $TrA \models \neg(X, TrC, \alpha)$  if there is no injective monomorphism  $h : TrC \rightarrow TrA$  such that  $D \models_h \alpha$ .

Then we could define the semantics of a transformation specification  $TSP$  as the class of all triple algebras that satisfy the patterns in  $TSP$ . However, in our opinion, as argued in [23], this semantics would be too loose. In particular, it would include triple algebras whose target and connection part cannot be *generated* by the patterns. For instance, models whose target part includes some kinds of elements of a given type not mentioned in the patterns. We think that restricting our attention to this kind of *generated models* is reasonable in this context. This is similar to the “No Junk” condition in algebraic specification.

**Definition 10 (TSP-generated triple algebras).** Given a pattern specification  $TSP$ , a triple algebra  $TrA$  is  $TSP$ -generated if there is a finite family of transformation patterns  $\{TP_k\}_{k \in K}$ , with  $TP_k = \langle X_k, \{TrC_k \xrightarrow{n_{jk}} N_{jk}\}_{jk \in J_k}, TrC_k, \alpha_k \rangle$ , and a family of monomorphisms  $\{TrC_k \xrightarrow{f_k} TrA\}_{k \in K}$  such that every  $f_k$  forward satisfies all the preconditions in  $N_k$ ,  $D \models_{f_k} \alpha_k$ , and  $f_1, \dots, f_n$  are jointly surjective.

Then, we can finally define the semantics of a transformation specification  $TSP$ :

$$Sem(TSP) = \{TrA \mid \forall P \in TSP \ TrA \models P \text{ and } TrA \text{ is a } TSP\text{-generated triple algebra}\}$$

## 5 Verification of Model Transformations

In [2] a formal framework for the verification of model transformations is proposed considering that the source and target models are, in some sense, semantically equivalent. This may be reasonable for some kinds of model transformations, like refactoring [26], or the example considered in [2] (again, the class diagrams-relational schemas transformation). However, this is not so reasonable in other classes of transformations: consider for instance (as an extreme case) a transformation from sequence diagrams

into class diagrams. In our opinion, a transformation may be considered correct if some observable properties of the source model remain (in some sense) invariant in the target model. For instance, in the sequence diagrams-class diagrams transformation we may only want to preserve typing. In the class diagrams-relational schemas transformation we may want to preserve the class-attribute association (including the association with inherited attributes). And in the case of some transformation between behaviour models we may want to preserve some behaviour properties like deadlock-freeness. This is the case when the main reason for applying the transformation is to use some verification tools which are available only for the class of target models.

As a consequence, we believe that the verification of a model transformation is only possible if the properties that characterize its correctness are given *a priori*. However, it may be not straightforward to state that a certain property remains invariant after a given transformation. The reason is that the source and target models may be quite different, so the corresponding properties may also look quite different. In this sense, we believe that the use of triple algebras and triple patterns to model transformations may help. In particular, we think that the possibility of dealing together with the source and target models and the connections between them helps in expressing simultaneously that a given property holds in the source model if and only if the corresponding property holds in the target model. For instance, the positive algebraic pattern presented in Example 1 can be seen as expressing that the class-attribute association remains invariant in the class diagrams-relational schemas transformation. This means that to verify a model transformation we must first provide a *verification specification*. More precisely, verification specifications should consist only of positive patterns. There are two reasons for this. On one hand, typically properties expressing invariants are, in this context, positive properties. On the other, negative patterns play a better role in transformation specifications, describing what kind of situations are forbidden. Actually, if the model transformation is going to be implemented by graph transformation (or a similar technique) as in [23] then the negative patterns may be embedded into the transformation rules as negative application conditions. Then a model transformation specified by *TSP* would be correct with respect to a verification specification *VSP* if every triple algebra in the semantics of *TSP* satisfies *VSP*:

**Definition 11 (Verification specification, Correctness of transformation specifications).** A *verification specification* *VSP* is a finite set of triple algebraic patterns. Then a transformation specification *TSP* is correct with respect to *VSP* if for every triple algebra  $TrA \in Sem(TSP)$  and every triple  $TrP \in VSP$   $TrA \models TrP$ .

Now, the next question is, given specifications *TSP* and *VSP*, how can we verify that *TSP* is correct with respect to *VSP*? If we have a method to generate all the triple algebras in the semantics of *TSP*, as in [23], an obvious, (but not very efficient) way would be to generate all these triple algebras and check that they satisfy the verification conditions. This approach may seem not very elegant but one could devise a deductive method based on these ideas. However, the main problem with this kind of solution is that a procedure based on this approach would only be a semi-decision procedure that would provide an answer in finite time only if *TSP* is incorrect, since normally there would be an infinite number of triple algebras in *Sem(TSP)*.

Instead, in what follows, we describe the basis of a finitely terminating procedure that, if the answer is positive, ensures that a given transformation specification is correct with respect to  $VSP$ . The basic idea is quite simple. Since the triple algebras in  $Sem(TSP)$  are  $TSP$ -generated, this means that each triple algebra in  $Sem(TSP)$  can be seen as the gluing of some of the transformation patterns in  $TSP$ . This means that we can be sure that  $TSP$  is correct with respect to  $VSP$ , if we are able to prove that each possible gluing of the transformation patterns in  $TSP$ , which satisfies the negative patterns in  $TSP$ , also satisfies the patterns in  $VSP$ . In particular, the key aspect is that, even if the number of possible ways of gluing the patterns in  $TSP$  is also infinite, it is enough to check a finite number of them to ensure correctness. In particular, it is enough to check the gluings which are *minimal*. But before describing this technique we first need some auxiliary definitions. The first one describes a notion of satisfaction between triple patterns (more precisely, satisfaction of arbitrary triple patterns by basic patterns). The definition is quite close to the definition of satisfaction of triple patterns by triple algebras. The main difference is on how we deal with the formulas involved in the patterns.

**Definition 12 (Satisfaction of triple patterns by basic patterns).** A basic triple algebraic pattern  $P = (X, TrC, \alpha)$  satisfies a conditional pattern  $CP = (X_1, X_2, c : TrC_1 \rightarrow TrC_2, \alpha_1, \alpha_2)$ , where  $X$  and  $X_1 \cup X_2$  are disjoint, denoted  $P \models CP$ , if for each triple monomorphism  $m : TrC_1 \rightarrow TrC$  such that  $\alpha \wedge \alpha_1 \wedge eq(m)$  is satisfiable in  $D$  there is a triple monomorphism  $m' : TrC_2 \rightarrow TrC$  such that  $m = m' \circ c$  and  $\alpha \wedge \alpha_2 \wedge eq(m')$  is satisfiable in  $D$ , where  $eq(m)$  and  $eq(m')$  denote, respectively, conjunctions of equations  $x = m(x)$  and  $x = m'(x)$ , for each  $x$  in  $X$ , and where  $m(x)$  denotes the image of  $x$  in  $X_1$  through  $m$  and, similarly,  $m'(x)$  denotes the image of  $x$  in  $X_2$  through  $m'$ .

The second definition that we need is the notion of gluing of patterns.

**Definition 13 (Gluing of patterns).** A basic triple algebraic pattern  $P = (X, TrC, \alpha)$  is the gluing of the finite family of transformation patterns  $\{TP_k\}_{k \in K}$ , with  $TP_k = \langle X_k, \{TrC_k \xrightarrow{n_{jk}} N_{jk}\}_{jk \in J_k}, TrC_k, \alpha_k \rangle$ , where all the  $X_k$  are pairwise disjoint, via a family of monomorphisms  $\{TrC_k \xrightarrow{f_k} TrC\}_{k \in K}$  if every  $f_k$  satisfies all the preconditions in  $\mathcal{N}_k$ ,  $f_1, \dots, f_n$  are jointly surjective, and  $\alpha = \bigwedge_k \alpha_k \wedge eq(\{f_k\}_{k \in K})$ , where  $eq(\{f_k\}_{k \in K})$  denotes the conjunction of equations  $x_i = x_j$  with  $x_i \in X_i$ ,  $x_j \in X_j$  and  $f_i(x_i) = f_j(x_j)$ .

The last definition that we need is the definition of minimal gluing. The idea is that a  $TSP$ -generated pattern  $P = (X, TrC, \alpha_1)$  is minimal with respect to a monomorphism  $h : TrC' \rightarrow TrC$  if all the patterns that participate in the gluing have some element in the image of  $h$  and if, in addition, if the elements in the image of  $h$  covered by a pattern are not covered already by the other patterns.

**Definition 14 (Minimal gluing of patterns).** A basic triple algebraic pattern  $P = (X, TrC, \alpha)$  which is the gluing of the family  $\{TP_k\}_{k \in K}$  via a family of monomorphisms  $\{TrC_k \xrightarrow{f_k} TrA\}_{k \in K}$  is minimal with respect to the monomorphism  $h : TrC_0 \rightarrow TrC$  if there is no pattern  $P' = (X', TrC', \alpha')$  which is the gluing of the family  $\{TP_j\}_{j \in J}$  via  $\{TrC_j \xrightarrow{f'_j} TrC\}_{j \in J}$ , with  $J \subset K$ , such that there are two monomorphisms  $m : TrC_0 \rightarrow TrC'$  and  $m' : TrC' \rightarrow TrC$  such that for every  $i \in J \cap K$ ,  $f'_i = m' \circ f_i$ , and moreover  $m' \circ m = h$ .

Now, we are ready to present the two results that are the basis for a procedure for checking the correctness of a transformation specification. The first result ensures the termination of the procedure:

**Theorem 1.** *Given a finite transformation specification  $TSP$ , and given a finite triple algebra with variables  $TrC_0$ , there is a finite number of monomorphisms  $h : TrC_0 \rightarrow TrC$  such that there is a pattern  $P = (X, TrC, \alpha)$  which is the gluing of a family of patterns in  $TSP$  and which is minimal with respect to  $h$ .*

*Proof.* Since  $TrC_0$  is finite, it has a finite number of elements, let us say  $n$ . This means that if  $P$  is minimal with respect to a monomorphism  $h$ , then  $P$  is the gluing of at most  $n$  patterns. In addition, we know that if  $TP = \langle X, \{TrC' \xrightarrow{n_j} N_j\}_{j \in J}, TrC', \alpha \rangle$  is a pattern in  $TSP$  then  $TrC'$  is a finite algebra. This means that there is a finite number of patterns  $P$  which are the gluing of  $n$  or less patterns in  $TSP$ . Finally, since  $TrC_0$  and  $TrC$  are both finite, there is a finite number of monomorphisms from  $TrC_0$  into  $TrC$ . ■

The second result states that if all the minimal patterns obtained by overlapping transformation patterns in  $TSP$  satisfy a verification pattern then all triple algebras in  $Sem(TSP)$  also satisfy the verification pattern.

**Theorem 2.** *Given a finite transformation specification  $TSP$ , if for every triple pattern  $CP = (X_1, X_2, c : TrC_1 \rightarrow TrC_2, \alpha_1, \alpha_2)$  in the verification specification  $VSP$ , every monomorphism  $h : TrC_1 \rightarrow TrC$  such that there is a pattern  $P = (X, TrC, \alpha)$  which is the gluing of a family of patterns in  $TSP$  and which is minimal with respect to  $h$  and such that  $TrC$  satisfies the negative constraints in  $TSP$  we have that if  $P \models CP$  then  $TSP$  is correct with respect to  $VSP$ .*

*Proof.* (Sketch)

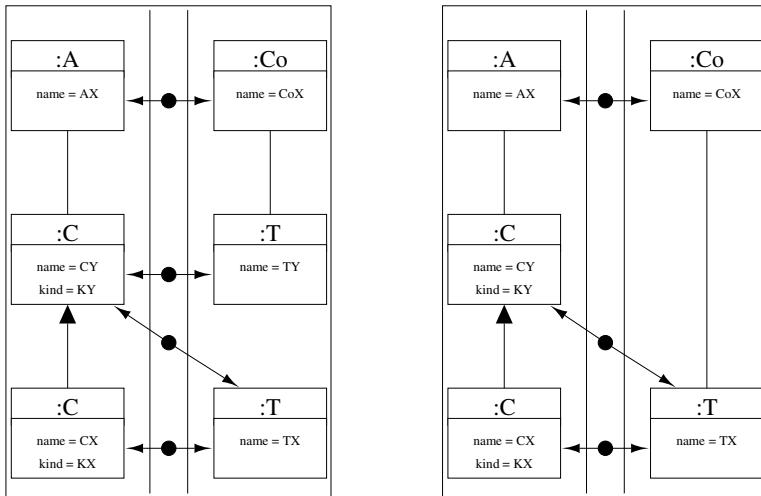
Let us suppose that  $TSP$  is not correct with respect to  $VSP$ . This means that there is a triple algebra  $TrA$  in  $Sem(TSP)$  that does not satisfy some triple pattern  $CP = (X_1, X_2, c : TrC_1 \rightarrow TrC_2, \alpha_1, \alpha_2)$  in  $VSP$ . Since  $TrA \in Sem(TSP)$  then  $TrA$  is  $TSP$  – generated. This implies that there is a finite family of transformation patterns  $\{TP_k\}_{k \in K}$ , with  $TP_k = \langle X_k, \{TrC_k \xrightarrow{n_{jk}} N_j\}_{j \in J_k}, TrC_k, \alpha_k \rangle$ , and a family of monomorphisms  $\{TrC_k \xrightarrow{f_k} TrA\}_{k \in K}$  such that every  $f_k$  forward satisfies all the preconditions in  $\mathcal{N}_k$ ,  $D \models_{f_k} \alpha_k$ , and  $f_1, \dots, f_n$  are jointly surjective. But then we can build a pattern  $P = (X, TrC, \alpha)$  by gluing the family  $\{TP_k\}_{k \in K}$  via some monomorphisms  $\{TrC_k \xrightarrow{f_k} TrC\}_{k \in K}$  in such a way that there is a monomorphism  $g : TrC \rightarrow TrA$ . Moreover, if  $TrA$  satisfies all the negative constraints in  $SP$ , then  $P$  also satisfies all the negative constraints in  $TSP$  and if  $TrA$  does not satisfy  $CP$  then  $P$  does not satisfy  $CP$  either.

Now, if  $P$  does not satisfy  $CP$  this means that there is a monomorphism  $m : TrC_1 \rightarrow TrC$  such that  $\alpha \wedge \alpha_1 \wedge eq(m)$  is satisfiable in  $D$ , but there is no monomorphism  $m' : TrC_2 \rightarrow TrC$  such that  $m = m' \circ c$  and  $\alpha \wedge \alpha_2 \wedge eq(m')$  is satisfiable in  $D$ . Now, if  $P$  is minimal with respect to  $m$  we have proved the theorem. If  $P$  is not minimal with respect to  $h$  then let  $P' = (X', TrC', \alpha')$  be obtained by the gluing of a family of  $\{TP_j\}_{j \in J}$ , with  $J \subset K$  via  $\{TrC_j \xrightarrow{f_j} TrC\}_{j \in J}$  and such that there are monomorphisms  $h : TrC_0 \rightarrow TrC'$

and  $h' : TrC' \rightarrow TrC$  such that for every  $i \in J \cap K$ ,  $f'_i = h \circ f_i$ , and moreover  $h' \circ h = m$ . Then it is straightforward to see  $P'$  satisfies all the negative constraints in  $TSP$  and does not satisfy  $CP$ . ■

*Example 3.* Let us now prove that the transformation from class diagrams to relational schemas consisting of the transformation patterns presented in Example 2 together with the negative pattern presented in Example 1 satisfies the conditional pattern in Example 1 (let us call it *Attribute inheritance*). First of all, we can see that the condition in that pattern cannot be matched to the postcondition of any of the transformation patterns in Example 2.

In this case, the only minimal patterns that can match the condition of the *Attribute inheritance* pattern are obtained by gluing the transformation patterns (1) and (2) and are displayed below:



with  $KX = \text{persistent} \wedge$   
 $\wedge KY = \text{Persistent} \wedge$   
 $\wedge CoX = " - " + AX$

with  $KX = \text{persistent} \wedge$   
 $\wedge KY = \text{Persistent} \wedge$   
 $\wedge CoX = " - " + AX$

Now, the pattern on the left violates the negative pattern that specifies that one class cannot have two associated tables. Therefore, we can discard it. Then, pattern on the right is the only minimal gluing that satisfy the negative condition on the transformation specification. But this pattern satisfies the *Attribute inheritance* pattern. Therefore according to Theorem 2 the transformation specification is correct with respect to the *Attribute inheritance* pattern.

## 6 Conclusion and Future Work

In this paper we have discussed what does it mean to verify a model transformation, and we have proposed a specific method for doing so. More precisely, first, we have presented a method for specifying model transformations that can be considered a generalization of the method presented in [6,12] and, then, we have presented a method to prove the correctness of a model transformation specification.

There are two main lines of work that should be pursued in the future. The first one has to do with refining and implementing the work presented in this paper. In particular, before implementing these ideas, it would be needed to present a specific algorithm for checking the correctness of a transformation based on the result presented in Theorem 2. Then, for the implementation there are two possibilities. On one hand, we could try to implement this approach using a tool like Maude [5] that would allow us to directly work on algebras. Or, alternatively, we could specialize our approach to the case of graphs, i.e. as in [6,12], and use some graph transformation tool.

The second line of work has to do with the kind of models considered. As said above, in this work we consider only the transformation of structural models. This means that the ideas presented here may have limited interest for the verification of transformation of behavioural models. In that case, one is not so much interested in knowing that each instance of the source model is correctly transformed into an instance of the target model, but that some observable properties of the overall behaviour of the source model remain invariant are also satisfied by the target model. This makes the problem technically very different.

## References

1. Boronat, A., Heckel, R., Meseguer, J.: Rewriting logic semantics and verification of model transformations. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 18–33. Springer, Heidelberg (2009)
2. Boronat, A., Knapp, A., Meseguer, J., Wirsing, M.: What is a multi-modelling language? In: WADT 2008 (2008)
3. Boronat, A., Meseguer, J.: An algebraic semantics for MOF. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 377–391. Springer, Heidelberg (2008)
4. Bouhoula, A., Jouannaud, J.-P., Meseguer, J.: Specification and proof in membership equational logic. *Theor. Comput. Sci.* 236(1-2), 35–132 (2000)
5. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C. (eds.): All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
6. de Lara, J., Guerra, E.: Pattern-based model-to-model transformation. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 426–441. Springer, Heidelberg (2008)
7. Ehrig, H., Baldamus, M., Orejas, F.: Amalgamation and extension in the framework of specification logics and generalized morphisms. *Bulletin of the EATCS* 44, 129–143 (1991)
8. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of algebraic graph transformation. Springer, Heidelberg (2006)
9. Ehrig, H., Habel, A.: Graph grammars with application conditions. In: Rozenberg, G., Salomaa, A. (eds.) *The Book of L*, pp. 87–100. Springer, Heidelberg (1986)

10. Goguen, J.A., Burstall, R.M.: Institutions: Abstract model theory for specification and programming. *J. ACM* 39(1), 95–146 (1992)
11. Goguen, J.A., Jouannaud, J.-P., Meseguer, J.: Operational semantics for order-sorted algebra. In: Brauer, W. (ed.) ICALP 1985. LNCS, vol. 194, pp. 221–231. Springer, Heidelberg (1985)
12. Guerra, E., de Lara, J., Orejas, F.: Pattern-based model-to-model transformation: Handling attribute conditions. In: ICMT 2009. LNCS. Springer, Heidelberg (accepted, 2009)
13. Habel, A., Heckel, R., Taentzer, G.: Graph grammars with negative application conditions. *Fundamenta Informatica* 287–313 (1996)
14. Habel, A., Pennemann, K.-H.: Correctness of high-level transformation systems relative to nested conditions. *Math. Struct. in Comp. Sc.* (to appear)
15. Heckel, R., Wagner, A.: Ensuring consistency of conditional graph rewriting - a constructive approach. ENTCS, 2 (1995)
16. Koch, M., Mancini, L.V., Parisi-Presicce, F.: Graph-based specification of access control policies. *J. Comput. Syst. Sci.*, 1–33 (2005)
17. Mens, T., Gorp, P.V.: A taxonomy of model transformation. *Electr. Notes Theor. Comput. Sci.* 152, 125–142 (2006)
18. Mosses, P.D.: Unified algebras. In: ADT (1988)
19. Mosses, P.D.: Unified algebras and institutions. In: LICS 1989, pp. 304–312 (1989)
20. Mosses, P.D.: Unified algebras and modules. In: Sixteenth Annual ACM Symposium on Principles of Programming Languages, POPL 1989, pp. 329–343 (1989)
21. Orejas, F.: Attributed graph constraints. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 274–288. Springer, Heidelberg (2008)
22. Orejas, F., Ehrig, H., Prange, U.: A logic of graph constraints. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 179–198. Springer, Heidelberg (2008)
23. Orejas, F., Guerra, E., de Lara, J., Ehrig, H.: Correctness, completeness and termination of pattern-based model-to-model transformation. In: Kurz, A., Lenisa, M., Tarlecki, A. (eds.) CALCO 2009. LNCS, vol. 5728, pp. 383–397. Springer, Heidelberg (2009)
24. Pennemann, K.-H.: Resolution-like theorem proving for high-level condition. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 289–304. Springer, Heidelberg (2008)
25. QVT (2005), <http://www.omg.org/docs/ptc/05-11-01.pdf>
26. Rangel, G., Lambers, L., König, B., Ehrig, H., Baldan, P.: Behavior preservation in model refactoring using DPO transformations with borrowed contexts. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 242–256. Springer, Heidelberg (2008)
27. Schürr, A.: Specification of graph translators with triple graph grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995)

# Towards Compatible and Interderivable Semantic Specifications for the Scheme Programming Language, Part I: Denotational Semantics, Natural Semantics, and Abstract Machines

Olivier Dany

Department of Computer Science, Aarhus University  
Aabogade 34, DK-8200 Aarhus N, Denmark  
`danny@cs.au.dk`

**Abstract.** We derive two big-step abstract machines, a natural semantics, and the valuation function of a denotational semantics based on the small-step abstract machine for Core Scheme presented by Clinger at PLDI'98. Starting from a functional implementation of this small-step abstract machine, (1) we fuse its transition function with its driver loop, obtaining the functional implementation of a big-step abstract machine; (2) we adjust this big-step abstract machine so that it is in de-functionalized form, obtaining the functional implementation of a second big-step abstract machine; (3) we refunctionalize this adjusted abstract machine, obtaining the functional implementation of a natural semantics in continuation-passing style; and (4) we closure-unconvert this natural semantics, obtaining a compositional continuation-passing evaluation function which we identify as the functional implementation of a denotational semantics in continuation-passing style. We then compare this valuation function with that of Clinger's original denotational semantics of Scheme.

## 1 Introduction

*Motivation:* Somewhat facetiously, in an earlier work [7], Biernacka and the author concluded:

Call/cc was introduced in Scheme [11] as a Church encoding of Reynolds's escape operator [49]. A typed version of it is available in Standard ML of New Jersey [30] and Griffin has identified its logical content [29]. It is endowed with a variety of specifications: a CPS transformation [18], a CPS interpreter [31, 49], a denotational semantics [34], a computational monad [56], a big-step operational semantics [30], the CEK machine [27], calculi in the form of reduction semantics [26], and a number of implementation techniques [12, 15, 32]—not to mention its call-by-name variant in the archival version of Krivine's machine [35].

Question: How do we know that all the artifacts<sup>1</sup> in this semantic jungle define the same call/cc?

In some sense, a similar story could be told about the Scheme programming language today, considering its independent specifications as a denotational semantics [47], a structural operational semantics [46] and a reduction semantics [38], and its tail-recursion property as accounted for with a small-step abstract machine [14]. Which one specifies the core of Scheme? The newest one? Or does one supersede a previous one of the same kind, just as the revised<sup>n+1</sup> report supersedes the revised<sup>n</sup> report [54, 53, 47, 13, 34, 51]? More likely, each semantics specifies one particular facet, however large that facet may be, of Scheme, irrespective of others. In that case, is the reference semantics the most complete one at any point of time?

*Background:* It is the author’s thesis<sup>2</sup> that functional representations of small-step operational semantics (i.e., structured operational semantics), reduction semantics (i.e., small-step operational semantics with an explicit representation of the reduction context), small-step abstract machines, big-step abstract machines, big-step operational semantics (i.e., natural semantics), and denotational semantics are inter-derivable using elementary program transformations such as CPS transformation [18, 45, 48, 52], defunctionalization [22, 48], fixed-point fusion [19, 44], and refocusing [23].

*This work:* As a proof of concept, we derive the functional representation of the denotational semantics corresponding to Clinger’s abstract machine as presented at PLDI’98 to specify the meaning of proper tail recursion [14]. We then compare it with Clinger’s denotational semantics in the R<sup>3</sup>RS [47]. Each of these semantics is significant: the denotational one was profoundly influential in that it revealed formal semantics in action to a whole generation of computer scientists, and the operational one was instrumental to precisely substantiate what it means for an implementation to be properly tail-recursive. They are also unique because of their use of permutation/unpermutation functions to account for the undetermined sequencing order of sub-expressions in an application.

Our starting point is that, as pioneered by Reynolds in “Definitional Interpreters” [48] and pursued by the author and his students [2, 3, 4, 6, 16], closure-converting and defunctionalizing a continuation-passing evaluation function yields a big-step abstract machine. Here, we restate Clinger’s machine to operate with big steps, we adjust it so that it is in defunctionalized form, and we refunctionalize and then closure-unconvert the functional implementation of this adjusted machine, obtaining an evaluation function which is compositional and in continuation-passing style.

---

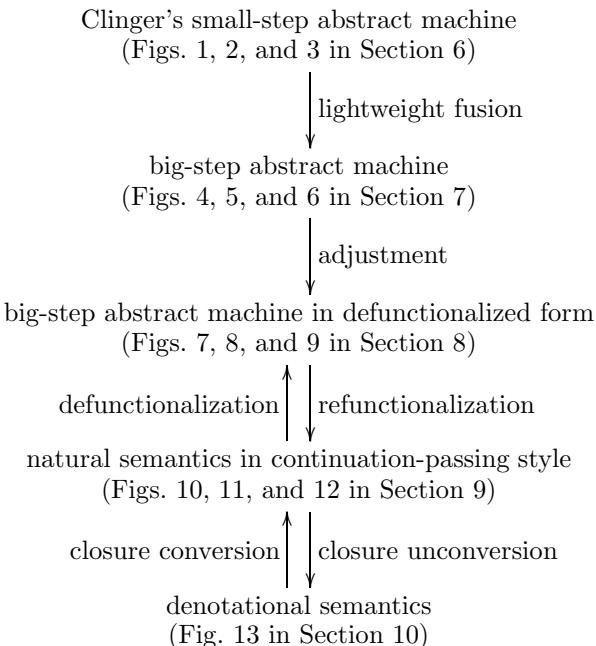
<sup>1</sup> Here, “artifact” is meant as “man-made construct.”

<sup>2</sup> A real one, actually [17], albeit not the only one [1, 5, 9, 33, 40, 39, 42, 43].

*Prerequisites:* Naturally, we assume the reader to be aware of Clinger’s denotational semantics of Scheme in the R<sup>3</sup>RS [47] and of his abstract machine for Core Scheme as presented at PLDI’98 [14]. In addition, we also expect a basic knowledge of Standard ML [41], a pure subset of which we use as a functional meta-language, and a passing familiarity with defunctionalization [22,48] and its left inverse, refunctionalization [21] as well as with closure conversion [36] and its left inverse, closure unconversion. As for the technique of fusing the transition function of a small-step abstract machine with its driver loop, it is described in a recent note by Millikin and the author [20].

*Overview:* We first review the domain of discourse (Section 2): stores, environments, and the permutation/unpermutation functions that are idiosyncratic to Clinger’s semantic specifications of Scheme. We then specify the syntax (Section 3) and the semantics (Section 4) of Core Scheme, and a garbage-collection rule (Section 5). Thus equipped, we present Clinger’s small-step abstract machine (Section 6) and then its big-step counterpart (Section 7). We put this big-step counterpart in defunctionalized form (Section 8), we present its refunctionalized counterpart (Section 9), and we closure-unconvert it (Section 10). We then compare the resulting compositional evaluation function in continuation-passing style to the valuation function of Clinger’s denotational semantics (Section 11) and then conclude (Section 12).

Pictorially:



## 2 Domain of Discourse

In the interest of brevity and abstractness, we only present ML signatures for the store (Section 2.1) and the environment (Section 2.2). We also explicitly treat Clinger's permutations and unpermutations (Section 2.3).

### 2.1 Store

A store is a mapping from locations to storable values. We specify it as a polymorphic abstract data type with the usual algebraic operators to allocate fresh locations and initialize them with given storable values, dereference a given location in a given store, and update a given store at a given location with a given storable value.

```
signature STO = sig
  type 'a sto
  type loc

  val empty : 'a sto

  val new : 'a sto * 'a      -> loc      * 'a sto
  val news : 'a sto * 'a list -> loc list * 'a sto

  val fetch : loc *      'a sto -> 'a option
  val update : loc * 'a * 'a sto -> 'a sto option
end

structure Sto :> STO = struct
  (* deliberately omitted *)
end
```

At the price of modularity, we could define the allocation operators to also be given an environment and a context so that the fresh location(s) they return can be more precisely characterized as not occurring in the environment and in the context. This less modular definition would let us implement Clinger's small-step abstract machine even more faithfully.

### 2.2 Environment

An environment is a mapping from identifiers to denotable values. We specify it as a polymorphic abstract data type with the usual algebraic operators to extend a given environment with new bindings and to look up identifiers in a given environment. We also need a predicate testing whether a given environment is empty.

```
type ide = string

signature ENV = sig
  type 'a env

  val empty : 'a env
  val emptyp : 'a env -> bool
```

```

val extend : ide * 'a * 'a env -> 'a env
val extends : ide list * 'a list * 'a env -> 'a env

val lookup : ide * 'a env -> 'a option
end

structure Env :> ENV = struct
  (* deliberately omitted *)
end

```

## 2.3 Permutations and Unpermutations

Both in his denotational semantics and his small-step abstract machine, Clinger non-deterministically uses a pair of permutation functions: one over the subterms in an application, and the inverse one over the resulting values (see the note in Section 6.1). We implement this non-determinism by threading a stream of pairs of permutations through Clinger's semantic specifications. We materialize this stream with the following polymorphic abstract data type.

```

signature PERM = sig
  type 'a perm
  type ('v, 't) permgen

  val init : ('v, 't) permgen
  val new : ('v, 't) permgen -> ('v perm * 't perm) * ('v, 't) permgen
end

structure Perm :> PERM = struct
  (* deliberately omitted *)
end

```

## 3 Syntax

The following module implements the internal syntax of Core Scheme [14, Fig. 1]. Literals are represented using the quotation mechanism from Lisp. Variables are either predeclared or are given rise as formal parameters of a lambda-abstraction. Lambda-abstractions and applications are uncurried (and with a fixed arity). Conditional expressions are as in Lisp. Any variable can be assigned.

```

structure Syn = struct
  datatype quotation = QBOOL of bool
    | QNUMB of int
    | QSYMB of ide
    | QPAIR of Sto.loc * Sto.loc
    (* / QVECT of ... *)
    (* / ... *)

  datatype term = QUOTE of quotation
    | VAR of ide
    | LAM of ide list * term
    | APP of term * term list
    | COND of term * term * term
    | SET of ide * term
end

```

## 4 Semantics

The following module implements the expressible values and the evaluation contexts [14, Fig. 4] as well as the denotable values, which are store locations, and the storable values, which are expressible values.

```

structure Sem = struct
  type env = Sto.loc Env.env

  datatype primop = CWCC (* / ... *)

  datatype value =
    QUOTED of Syn.quotation
  | UNSPECIFIED
  | UNDEFINED
  | CLOSURE of (ide list * Syn.term) * env * Sto.loc
  | PRIMOP of primop * Sto.loc
  | ESCAPE of cont * Sto.loc
  and cont =
    HALT
  | SELECT of Syn.term * Syn.term * env * cont
  | ASSIGN of ide * env * cont
  | PUSH of Syn.term list * env * value list * value Perm.perm * cont
  | CALL of value list * cont

  type sto = value Sto.sto
end

```

Evaluating a quoted expression yields a quotation (tagged with QUOTED). Evaluating an assignment yields an unspecified value (tagged with UNSPECIFIED). Evaluating a lambda-abstraction yields a user-defined procedure (tagged with CLOSURE). The value of a primitive operator is a predefined procedure (tagged with PRIMOP). Like Clinger, we focus on Core Scheme but in addition, we also consider one predefined procedure, call/cc, and the subsequent escape procedures, i.e., first-class continuations (tagged with ESCAPE).

Evaluation contexts are inductively constructed through the data type cont. A context is either empty (and tagged with HALT), or the context of the test subterm in a conditional expression (it is then tagged with SELECT), of the subterm in an assignment (it is then tagged with ASSIGN), of a subterm in an application (it is then tagged with PUSH), or of a value in an application where all the other subcomponents are already evaluated (it is then tagged with CALL).

## 5 Garbage Collection

The following module implements the garbage-collection rule [14, Fig. 5].

```

signature GC = sig
  val gc : Sem.value * Sem.env * Sem.cont * Sem.sto -> Sem.sto
end

structure Gc :> GC = struct
  (* deliberately omitted *)
end

```

---

```

structure M_small_step = struct
  local val (l_cwcc, s1) = Sto.new (Sto.empty, Sem.UNSPECIFIED)
    val (l, s2) = Sto.new (s1, Sem.PRIMOP (Sem.CWCC, l_cwcc))
  in val env_init = Env.extend ("call/cc", l, Env.empty)
    val sto_init = s2
  end

  type env = Sem.env
  type sto = Sem.sto

  type perms = (Sem.value, Syn.term) Perm.permgen

  datatype term_or_value = TERM of Syn.term
    | VALUE of Sem.value

  datatype halting_state = RESULT of Sem.value * sto * perms
    | STUCK of string

  datatype state = FINAL of halting_state
    | INTER of configuration
  withtype configuration = term_or_value * env * Sem.cont * sto * perms

  (* move : term_or_value * env * Sem.cont * sto * perms -> state *)
  fun move ... =
    (* see Figs. 2 & 3 *)

  (* drive : state -> halting_state *)
  fun drive (FINAL answer) =
    answer
  | drive (INTER configuration) =
    drive (move configuration)

  (* evaluate : Syn.term -> halting_state *)
  fun evaluate t =
    drive (INTER (TERM t, env_init, Sem.HALT, sto_init, Perm.init))
end

```

---

**Fig. 1.** Clinger's small-step abstract machine without the GC rule, part 1/3:configurations, driver loop and initialization

## 6 Clinger's Small-Step Abstract Machine

We first present the machine without the GC rule (Section 6.1) and then with a GC rule (Section 6.2).

### 6.1 Without the GC Rule

The abstract machine is displayed in Figs. 1, 2, and 3. It operates on a quintuple: a term or a value, an environment mapping variables to store locations, a context, a store mapping locations to values, and a stream of permutations. When the first component is a term, this term is dispatched upon (see Fig. 2). When the first component is a value, the third component (i.e., the context) is dispatched upon (see Fig. 3).

---

```

fun move (TERM (Syn.QUOTE q), r, ec, s, pg) =
  INTER (VALUE (Sem.QUOTED q), r, ec, s, pg)
| move (TERM (Syn.VAR i), r, ec, s, pg) =
  (case Env.lookup (i, r)
   of (SOME l)
     => (case Sto.fetch (l, s)
      of (SOME v)
        => (case v
         of Sem.UNDEFINED
           => FINAL (STUCK "undefined value")
          | _
            => INTER (VALUE v, r, ec, s, pg))
     | NONE
       => FINAL (STUCK "attempt to read an invalid location"))
   | NONE
     => FINAL (STUCK "attempt to reference an undeclared variable"))
| move (TERM (Syn.LAM (is, t)), r, ec, s, pg) =
  let val (l, s') = Sto.new (s, Sem.UNSPECIFIED)
  in INTER (VALUE (Sem.CLOSURE ((is, t), r, l)), r, ec, s', pg)
  end
| move (TERM (Syn.COND (t0, t1, t2)), r, ec, s, pg) =
  INTER (TERM t0, r, Sem.SELECT (t1, t2, r, ec), s, pg)
| move (TERM (Syn.SET (i, t)), r, ec, s, pg) =
  INTER (TERM t, r, Sem.ASSIGN (i, r, ec), s, pg)
| move (TERM (Syn.APP (t0, ts)), r, ec, s, pg) =
  let val ((pi, rev_pi_inv), pg') = Perm.new pg
    val (t0', ts') = rev_pi_inv (t0, ts)
  in INTER (TERM t0', r, Sem.PUSH (ts', r, nil, pi, ec), s, pg')
  end
| move (VALUE v, r, ec, s, pg) =
  (* see Fig. 3 *)

```

---

**Fig. 2.** Clinger’s small-step abstract machine without the GC rule, part 2/3:transition function over terms

**Note:** A new pair of permutations is explicitly allocated every time an application is evaluated. The second one operates over terms and is used immediately over the sub-terms of the application. The first one operates over values and it will be subsequently used over the results of evaluating each of these sub-terms.

Clinger’s formulation uses an infinite set of rules generated by a rule schema that is parameterized by a permutation and its inverse. We model his non-deterministic choice of permutation with an oracle that picks in the current stream of permutations.

For the rest, Figs. 1, 2, and 3 display a scrupulously faithful implementation of Clinger’s small-step abstract machine, including its penultimate transition.

## 6.2 With a GC Rule

The following implementation deterministically applies the GC rule from Section 5 every time a value is returned to the evaluation context, i.e., at every reduction step. The garbage-collection function is passed the current roots and

---

```

fun move (TERM t, r, ec, s, pg) =
  (* see Fig. 2 *)
  | move (VALUE v, r', Sem.HALT, s, pg) =
    if Env.empty(r'
      then FINAL (RESULT (v, s, pg))
      else INTER (VALUE v, Env.empty, Sem.HALT, s, pg)
  | move (VALUE (Sem.QUOTED (Syn.QBOOL false)), r', Sem.SELECT (t1, t2, r, ec), s, pg) =
    INTER (TERM t2, r, ec, s, pg)
  | move (VALUE _, r', Sem.SELECT (t1, t2, r, ec), s, pg) =
    INTER (TERM t1, r, ec, s, pg)
  | move (VALUE v, r', Sem.ASSIGN (i, r, ec), s, pg) =
    (case Env.lookup (i, r)
      of (SOME l)
        => (case Sto.update (l, v, s)
          of (SOME s')
            => INTER (VALUE Sem.UNSPECIFIED, r, ec, s', pg)
          | NONE
            => FINAL (STUCK "attempt to write an invalid location"))
        | NONE
          => FINAL (STUCK "attempt to assign an undeclared variable"))
  | move (VALUE v0', r', Sem.PUSH (nil, r, vs', pi, ec), s, pg) =
    let val (v0', vs') = pi (v0', vs')
    in INTER (VALUE v0, r, Sem.CALL (vs, ec), s, pg)
    end
  | move (VALUE v0', r', Sem.PUSH (t1' :: ts', r, vs', pi, ec), s, pg) =
    INTER (TERM t1', r, Sem.PUSH (ts', r, v0' :: vs', pi, ec), s, pg)
  | move (VALUE (Sem.CLOSURE ((is, t), r, l)), r', Sem.CALL (vs, ec), s, pg) =
    if List.length vs = List.length is
    then let val (ls, s') = Sto.news (s, vs)
      in INTER (TERM t, Env.extends (is, ls, r), ec, s', pg)
      end
    else FINAL (STUCK "arity mismatch")
  | move (VALUE (Sem.PRIMOP (Sem.CWCC, _)), r', Sem.CALL (vs, ec), s, pg) =
    if List.length vs = 1
    then let val (l, s') = Sto.new (s, Sem.UNSPECIFIED)
      in INTER (VALUE (hd vs), r', Sem.CALL ([Sem.ESCAPE (ec, l)], ec), s', pg)
      end
    else FINAL (STUCK "arity mismatch")
  | move (VALUE (Sem.ESCAPE (ec', l)), r', Sem.CALL (vs, ec), s, pg) =
    if List.length vs = 1
    then INTER (VALUE (hd vs), r', ec', s, pg)
    else FINAL (STUCK "arity mismatch")
  | move (VALUE _, r', Sem.CALL (vs, ec), s, pg) =
    FINAL (STUCK "attempt to apply a non-procedure")

```

---

**Fig. 3.** Clinger's small-step abstract machine without the GC rule, part 3/3:transition function over values

yields a store that is used to continue evaluation. Compared to Fig. 1, only the driver loop is changed.

```

structure M_small_step_with_gc_rule = struct
  ...
  fun drive (FINAL answer) =
    answer
    | drive (INTER (configuration as (TERM t, r, ec, s, p))) =
      drive (move configuration)
    | drive (INTER (configuration as (VALUE v, r, ec, s, p))) =
      let val s' = Gc.gc (v, r, ec, s)
      in drive (move (VALUE v, r, ec, s', p))
      end
  ...
end

```

In contrast, Clinger's formulation of the GC rule is non-deterministic.

---

```

structure M_big_step = struct
  local val (l_cwcc, s1) = Sto.new (Sto.empty, Sem.UNSPECIFIED)
    val (l, s2) = Sto.new (s1, Sem.PRIMOP (Sem.CWCC, l_cwcc))
  in val env_init = Env.extend ("call/cc", l, Env.empty)
    val sto_init = s2
  end

  type env = Sem.env
  type sto = Sem.sto

  type perms = (Sem.value, Syn.term) Perm.permgen

  datatype term_or_value = TERM of Syn.term
                           | VALUE of Sem.value

  datatype answer = RESULT of Sem.value * sto * perms
                   | STUCK of string

  (* iterate : Syn.term * env * Sem.cont * sto * perms -> answer *)
  fun iterate ... =
    (* see Figs. 5 & 6 *)

  (* evaluate : Syn.term -> answer *)
  fun evaluate t =
    iterate (TERM t, env_init, Sem.HALT, sto_init, Perm.init)
end

```

---

**Fig. 4.** Big-step counterpart of Fig. 1, part 1/3:configurations and initialization

## 7 Big-Step Version of Clinger's Abstract Machine

We first present the big-step version of the machine without any GC rule (Section 7.1) and then with a GC rule (Section 7.2).

### 7.1 Without the GC Rule

In Fig. 1, the `drive` function iteratively calls the `move` function until a final answer is obtained, if any. As pointed out by Millikin and the author [20], such small-step abstract machines are candidates for lightweight fusion by fixed-point promotion [44]: the composition of `drive` and `move` can be fused into an ‘`iterate`’ function where the outer recursive call to `drive` has been distributed to all the return points in the definition of ‘`move`.’ The result is the big-step abstract machine displayed in Figs. 4, 5, and 6. Since Ohori and Sasano’s fixed-point promotion is fully correct, this big-step abstract machine is also correct, by construction.

---

```

fun iterate (TERM (Syn.QUOTE q), r, ec, s, pg) =
  iterate (VALUE (Sem.QUOTED q), r, ec, s, pg)
| iterate (TERM (Syn.VAR i), r, ec, s, pg) =
  (case Env.lookup (i, r)
   of (SOME l)
     => (case Sto.fetch (l, s)
          of (SOME sv)
            => (case sv
                  of (SOME v)
                    => iterate (VALUE v, r, ec, s, pg)
                  | NONE
                    => STUCK "attempt to reference an undefined variable")
            | NONE
              => STUCK "attempt to read an invalid location")
   | NONE
     => STUCK "attempt to reference an undeclared variable")
| iterate (TERM (Syn.LAM (is, t)), r, ec, s, pg) =
  let val (l, s') = Sto.new (s, Sem.UNSPECIFIED)
  in iterate (VALUE (Sem.CLOSURE ((is, t), r, l)), r, ec, s', pg)
  end
| iterate (TERM (Syn.COND (t0, t1, t2)), r, ec, s, pg) =
  iterate (TERM t0, r, Sem.SELECT (t1, t2, r, ec), s, pg)
| iterate (TERM (Syn.SET (i, t)), r, ec, s, pg) =
  iterate (TERM t, r, Sem.ASSIGN (i, r, ec), s, pg)
| iterate (TERM (Syn.APP (t0, ts)), r, ec, s, pg) =
  let val ((pi, rev_pi_inv), pg') = Perm.new pg
    val (t0', ts') = rev_pi_inv (t0, ts)
  in iterate (TERM t0', r, Sem.PUSH (ts', r, nil, pi, ec), s, pg')
  end
| iterate (VALUE ...) =
  (* see Fig. 6 *)

```

---

**Fig. 5.** Big-step counterpart of Fig. 2, part 2/3:transition function over terms

## 7.2 With a GC Rule

The following implementation deterministically applies the GC rule every time a function is about to be applied. Compared to Fig. 6, only one clause is changed.

```

structure M_big_step_with_gc_rule = struct
  (* ... *)
  | iterate (VALUE v0', r', Sem.PUSH (nil, r, vs', pi, ec), s, pg) =
    let val (v0, vs) = pi (v0', vs')
      val s' = Gc.gc (v0, r, ec, s)
    in iterate (VALUE v0, r, Sem.CALL (vs, ec), s', pg)
    end
  (* ... *)
end

```

---

```

fun iterate (TERM t, r, ec, s, pg) =
  (* see Fig. 5 *)
| iterate (VALUE v, r', Sem.HALT, s, pg) =
  if Env.emptyp r'
  then RESULT (v, s, pg)
  else iterate (VALUE v, Env.empty, Sem.HALT, s, pg)
| iterate (VALUE (Sem.QUOTED (Syn.QBOOL false)), r', Sem.SELECT (t1, t2, r, ec), s, pg) =
  iterate (TERM t2, r, ec, s, pg)
| iterate (VALUE _, r', Sem.SELECT (t1, t2, r, ec), s, pg) =
  iterate (TERM t1, r, ec, s, pg)
| iterate (VALUE v, r', Sem.ASSIGN (i, r, ec), s, pg) =
  (case Env.lookup (i, r)
   of (SOME l)
     => (case Sto.update (l, v, s)
          of (SOME s')
            => iterate (VALUE Sem.UNSPECIFIED, r, ec, s', pg)
          | NONE
            => STUCK "attempt to write an invalid location")
   | NONE
     => STUCK "attempt to assign an undeclared variable")
| iterate (VALUE v0', r', Sem.PUSH (nil, r, vs', pi, ec), s, pg) =
  let val (v0, vs) = pi (v0', vs')
  in iterate (VALUE v0, r, Sem.CALL (vs, ec), s, pg)
  end
| iterate (VALUE v0', r', Sem.PUSH (t1' :: ts', r, vs', pi, ec), s, pg) =
  iterate (TERM t1', r, Sem.PUSH (ts', r, v0' :: vs', pi, ec), s, pg)
| iterate (VALUE (Sem.CLOSURE ((is, t), r, l)), r', Sem.CALL (vs, ec), s, pg) =
  if List.length vs = List.length is
  then let val (ls, s') = Sto.news (s, vs)
    in iterate (TERM t, Env.extends (is, ls, r), ec, s', pg)
    end
  else STUCK "arity mismatch"
| iterate (VALUE (Sem.PRIMOP (Sem.CWCC, _)), r', Sem.CALL (vs, ec), s, pg) =
  if List.length vs = 1
  then let val (l, s') = Sto.new (s, Sem.UNSPECIFIED)
    in iterate (VALUE (hd vs), r', Sem.CALL ([Sem.ESCAPE (ec, l)], ec), s', pg)
    end
  else STUCK "arity mismatch"
| iterate (VALUE (Sem.ESCAPE (ec', l)), r', Sem.CALL (vs, ec), s, pg) =
  if List.length vs = 1
  then iterate (VALUE (hd vs), r', ec, s, pg)
  else STUCK "arity mismatch"
| iterate (VALUE _, r', Sem.CALL (vs, ec), s, pg) =
  STUCK "attempt to apply a non-procedure"

```

---

**Fig. 6.** Big-step counterpart of Fig. 2, part 3/3:transition function over values

---

```

structure M_big_step_defunct = struct
  local val (l_cwcc, s1) = Sto.new (Sto.empty, Sem.UNSPECIFIED)
    val (l, s2) = Sto.new (s1, Sem.PRIMOP (Sem.CWCC, l_cwcc))
  in val env_init = Env.extend ("call/cc", l, Env.empty)
    val sto_init = s2
  end

  type env = Sem.env
  type sto = Sem.sto

  type perms = (Sem.value, Syn.term) Perm.permgen

  datatype term_or_value = TERM of Syn.term
                        | VALUE of Sem.value

  datatype answer = RESULT of Sem.value * sto * perms
                  | STUCK of string

  (* eval : Syn.term * env * Sem.cont * sto * perms -> answer *)
  fun eval ... =
    (* see Fig. 8 *)
  (* continue : Sem.value * Sem.cont * sto * perms -> answer *)
  and continue ... =
    (* see Fig. 9 *)

  (* evaluate : Syn.term -> answer *)
  fun evaluate t =
    eval (t, env_init, Sem.HALT, sto_init, Perm.init)
end

```

---

**Fig. 7.** Version of Fig. 4 in defunctionalized form, part 1/3:configurations and initialization

## 8 Big-Step Version of Clinger’s Abstract Machine in Defunctionalized form

### 8.1 Without the GC rule

Like the SECD machine [16, 36], the big-step version of Clinger’s abstract machine is not in defunctionalized form. Fortunately, it can easily made to be so [21], by using the type isomorphism between the transition function

iterate : term\_or\_value \* env \* Sem.cont \* sto \* perms -> answer  
 and two mutually recursive transition functions

```

  eval : Syn.term * env * Sem.cont * sto * perms -> answer
  continue : Sem.value * Sem.cont * sto * perms -> answer

```

where we removed the dead environment parameter in `continue`.

The reformulated version is displayed in Figs. 7, 8, and 9, and can readily be recognized as an ‘eval/apply’ abstract machine [37]:<sup>3</sup> the ‘eval’ transition function dispatches on terms and the ‘apply’ transition function (or more accurately,

---

<sup>3</sup> A more accurate term than ‘eval/apply’, though, would be ‘eval/continuation’.

---

```

fun eval (Syn.QUOTE q, r, ec, s, pg) =
  continue (Sem.QUOTED q, ec, s, pg)
| eval (Syn.VAR i, r, ec, s, pg) =
  (case Env.lookup (i, r)
   of (SOME l)
     => (case Sto.fetch (l, s)
          of (SOME v)
            => (case v
                  of Sem.UNDEFINED
                    => STUCK "undefined value"
                  | _
                    => continue (v, ec, s, pg))
   | NONE
     => STUCK "attempt to read an invalid location")
   | NONE
     => STUCK "attempt to reference an undeclared variable")
| eval (Syn.LAM (is, t), r, ec, s, pg) =
  let val (l, s') = Sto.new (s, Sem.UNSPECIFIED)
  in continue (Sem.CLOSURE ((is, t), r, l), ec, s', pg)
  end
| eval (Syn.COND (t0, t1, t2), r, ec, s, pg) =
  eval (t0, r, Sem.SELECT (t1, t2, r, ec), s, pg)
| eval (Syn.SET (i, t), r, ec, s, pg) =
  eval (t, r, Sem.ASSIGN (i, r, ec), s, pg)
| eval (Syn.APP (t0, ts), r, ec, s, pg) =
  let val ((pi, rev_pi_inv), pg') = Perm.new pg
    val (t0', ts') = rev_pi_inv (t0, ts)
  in eval (t0', r, Sem.PUSH (ts', r, nil, pi, ec), s, pg')
  end
and continue (v, ec, s, pg) =
  (* see Fig. 9 *)

```

---

**Fig. 8.** Version of Figs. 5 and 6 in defunctionalized form, part 2/3: the transition function over terms

the ‘continue’ transition function) dispatches on (the top constructor of) the context. This abstract machine is in defunctionalized form in that *the evaluation context and the second transition function are the defunctionalized counterpart of a function*. As shown in the next section, this function is a continuation since the refunctionalized abstract machine is in continuation-passing style.<sup>4</sup>

## 8.2 With a GC Rule

As in Section 7.2, it is simple to deterministically apply the GC rule, e.g., every time a function is about to be applied.

---

<sup>4</sup> Hence the point about terminology in Footnote 3.

---

```

fun eval (t, r, ec, s, pg) =
  (* see Fig. 8 *)
and continue (v, Sem.HALT, s, pg) =
  RESULT (v, s, pg)
| continue (Sem.QUOTED (Syn.QBOOL false), Sem.SELECT (t1, t2, r, ec), s, pg) =
  eval (t2, r, ec, s, pg)
| continue (_, Sem.SELECT (t1, t2, r, ec), s, pg) =
  eval (t1, r, ec, s, pg)
| continue (v, Sem.ASSIGN (i, r, ec), s, pg) =
  (case Env.lookup (i, r)
   of (SOME l)
     => (case Sto.update (l, v, s)
          of (SOME s')
            => continue (Sem.UNSPECIFIED, ec, s', pg)
          | NONE
            => STUCK "attempt to write an invalid location")
   | NONE
     => STUCK "attempt to assign an undeclared variable")
| continue (v0', Sem.PUSH (nil, r, vs', pi, ec), s, pg) =
  let val (v0, vs) = pi (v0', vs')
  in continue (v0, Sem.CALL (vs, ec), s, pg)
  end
| continue (v0', Sem.PUSH (t1' :: ts', r, vs', pi, ec), s, pg) =
  eval (t1', r, Sem.PUSH (ts', r, v0' :: vs', pi, ec), s, pg)
| continue (Sem.CLOSURE ((is, t), r, l), Sem.CALL (vs, ec), s, pg) =
  if List.length vs = List.length is
  then let val (ls, s') = Sto.news (s, vs)
    in eval (t, Env.extends (is, ls, r), ec, s', pg)
  end
  else STUCK "arity mismatch"
| continue (Sem.PRIMOP (Sem.CWCC, _), Sem.CALL (vs, ec), s, pg) =
  if List.length vs = 1
  then let val (l, s') = Sto.new (s, Sem.UNSPECIFIED)
    in continue (hd vs, Sem.CALL ([Sem.ESCAPE (ec, l)], ec), s', pg)
  end
  else STUCK "arity mismatch"
| continue (Sem.ESCAPE (ec', l), Sem.CALL (vs, ec), s, pg) =
  if List.length vs = 1
  then continue (hd vs, ec', s, pg)
  else STUCK "arity mismatch"
| continue (_, Sem.CALL (vs, ec), s, pg) =
  STUCK "attempt to apply a non-procedure"

```

---

**Fig. 9.** Version of Figs. 5 and 6 in defunctionalized form, part 3/3: the transition function over contexts

## 9 Big-Step Version of Clinger's Abstract Machine, Refunctionalized

In Fig. 10, the key point is the definition of cont:

```
cont = value * sto * perms -> answer
```

Its definition as a data type in Section 4, together with the function continue in Fig. 9, were in defunctionalized form. The present definition takes the form of a function: instead of constructing an element of the data type of Section 4 and interpreting it in the function continue, a lambda-abstraction is created that, when applied, will carry out this interpretation.

---

```

structure Sem = struct
  type env = Sto.loc Env.env

  datatype primop = CWCC

  datatype value = QUOTED of Syn.quotation
    | UNSPECIFIED
    | UNDEFINED
    | CLOSURE of (ide list * Syn.term) * env * Sto.loc
    | PRIMOP of primop * Sto.loc
    | ESCAPE of cont * Sto.loc
  and   answer = RESULT of value * sto * perms
    | STUCK of string
  withtype perms = (value, Syn.term) Perm.permgen
  and   sto = value Sto.sto
  and   cont = value * sto * perms -> answer
end

structure M_big_step_refunct = struct
  local val (l_cwcc, s1) = Sto.new (Sto.empty, Sem.UNSPECIFIED)
    val (l, s2) = Sto.new (s1, Sem.PRIMOP (Sem.CWCC, l_cwcc))
  in val env_init = Env.extend ("call/cc", l, Env.empty)
    val sto_init = s2
  end

  type env = Sem.env
  type sto = Sem.sto

  type perms = (Sem.value, Syn.term) Perm.permgen

  datatype term_or_value = TERM of Syn.term
    | VALUE of Sem.value

  fun eval (t, r, k, s, pg) =
    (* see Fig. 11 *)
  and evalis (ts, v0', vs', pi, r, k, s, pg) =
    (* see Fig. 11 *)
  and apply (v0, vs, k, s, pg) =
    (* see Fig. 12 *)

  fun evaluate t =
    eval (t, env_init, fn (v, s, pg) => Sem.RESULT (v, s, pg), sto_init, Perm.init)
end

```

---

**Fig. 10.** Refunctionalized version of Fig. 7, part 1/3: configurations and initialization

## 9.1 Without the GC Rule

The refunctionalized version is displayed in Figs. 10, 11, and 12: defunctionalizing it yields back Figs. 7, 8, and 9. It is the evaluation function in continuation-passing style of a natural semantics [24,30]. As exploited in Section 10, it is also in the range of closure conversion.

## 9.2 With a GC Rule

Refunctionalization has made us cross a line: continuations are now higher-order, which prevents us to implement the GC rule as directly as in Section 7.2.

---

```

fun eval (Syn.QUOTE q, r, k, s, pg) =
  k (Sem.QUOTED q, s, pg)
| eval (Syn.VAR i, r, k, s, pg) =
  (case Env.lookup (i, r)
   of (SOME l)
     => (case Sto.fetch (l, s)
          of (SOME v)
            => (case v
                  of Sem.UNDEFINED
                    => Sem.STUCK "undefined value"
                  | _
                    => k (v, s, pg))
   | NONE
     => Sem.STUCK "attempt to read an invalid location")
  | NONE
    => Sem.STUCK "attempt to reference an undeclared variable")
| eval (Syn.LAM (is, t), r, k, s, pg) =
  let val (l, s') = Sto.new (s, Sem.UNSPECIFIED)
  in k (Sem.CLOSURE ((is, t), r, l), s', pg)
  end
| eval (Syn.COND (t0, t1, t2), r, k, s, pg) =
  eval (t0, r,
        fn (Sem.QUOTED (Syn.QBOOL false), s, pg)
          => eval (t2, r, k, s, pg)
        | (_, s, pg)
          => eval (t1, r, k, s, pg),
        s, pg)
| eval (Syn.SET (i, t), r, k, s, pg) =
  eval (t, r,
        fn (v, s, pg)
          => (case Env.lookup (i, r)
               of (SOME l)
                 => (case Sto.update (l, v, s)
                      of (SOME s')
                        => k (Sem.UNSPECIFIED, s', pg)
                      | NONE
                        => Sem.STUCK "attempt to write an invalid location")
                 | NONE
                   => Sem.STUCK "attempt to assign an undeclared variable"),
        s, pg)
| eval (Syn.APP (t0, ts), r, k, s, pg) =
  let val ((pi, rev_pi_inv), pg') = Perm.new pg
  val (t0', ts') = rev_pi_inv (t0, ts)
  in eval (t0', r,
            fn (v0', s, pg)
              => evlis (ts', v0', nil, pi, r, k, s, pg),
            s, pg')
  end
and evlis (nil, v0', vs', pi, r, k, s, pg) =
  let val (v0, vs) = pi (v0', vs')
  in apply (v0, vs, k, s, pg)
  end
| evals (t1' :: ts', v0', vs', pi, r, k, s, pg) =
  eval (t1', r,
        fn (v1', s, pg)
          => evlis (ts', v1', v0' :: vs', pi, r, k, s, pg),
        s, pg)
and apply (v0, vs, k, s, pg) =
  (* see Fig. 12 *)

```

---

**Fig. 11.** Refunctionalized version of Fig. 8, part 2/3: evaluation functions

---

```

fun eval (t, r, k, s, pg) =
  (* see Fig. 11 *)
and evlis (ts, v0', vs', pi, r, k, s, pg) =
  (* see Fig. 11 *)
and apply (Sem.CLOSURE ((is, t), r, _), vs, k, s, pg) =
  if List.length vs = List.length is
  then let val (ls, s') = Sto.news (s, vs)
    in eval (t, Env.extends (is, ls, r), k, s', pg)
  end
  else Sem.STUCK "arity mismatch"
| apply (Sem.PRIMOP (Sem.CWCC, _), vs, k, s, pg) =
  if List.length vs = 1
  then let val (l, s') = Sto.new (s, Sem.UNSPECIFIED)
    in apply (hd vs, [Sem.ESCAPE (k, l)], k, s', pg)
  end
  else Sem.STUCK "arity mismatch"
| apply (Sem.ESCAPE (k', _), vs, k, s, pg) =
  if List.length vs = 1
  then k' (hd vs, s, pg)
  else Sem.STUCK "arity mismatch"
| apply (_ , vs, k, s, pg) =
  Sem.STUCK "attempt to apply a non-procedure"

```

---

**Fig. 12.** Refunctionalized version of Fig. 8, part 3/3: procedure application

## 10 Big-Step Version of Clinger's Abstract Machine, Refunctionalized and Closure-Unconverted

So far all semantic artifacts represent three sorts of procedures: the user-defined ones, the predefined ones, and the escape ones. Closure unconversion lets us represent them as one function in the data type of values (see Fig. 13):

```
value list * cont * sto * perms -> answer
```

Each sort of procedure is now encoded as one such function and wrapped up in the PROCEDURE constructor together with a token store location.

### 10.1 Without the GC Rule

Fig. 13 displays the higher-order counterpart of Figs. 10, 11, and 12: closure-converting it yields back these three figures. It is a compositional evaluation function in continuation-passing style.

### 10.2 With a GC Rule

Closure unconversion has made us cross another line: the higher-order functions in the domain of values further prevent us from implementing the GC rule as directly as in Section 7.2.

---

```

structure Sem = struct
  ...
  datatype value = ...
    | PROCEDURE of (value list * cont * sto * perms -> answer) * Sto.loc
  ...
end

structure M_big_step_refunct_higher_order = struct
  fun cwcc (vs, k, s, pg) =
    if List.length vs = 1
    then let val (l, s') = Sto.new (s, Sem.UNSPECIFIED)
        in case hd vs
          of Sem.PROCEDURE (f, _) => f ([Sem.PROCEDURE (fn (vs, _, s, pg)
            => if List.length vs = 1
              then k (hd vs, s, pg)
              else Sem.STUCK "arity mismatch",
                1)],
                k, s', pg)
          | _ => Sem.STUCK "attempt to apply a non-procedure"
        end
      else Sem.STUCK "arity mismatch"

  local val (l_cwcc, s1) = Sto.new (Sto.empty, Sem.UNSPECIFIED)
    val (l, s2) = Sto.new (s1, Sem.PROCEDURE (cwcc, l_cwcc))
  in val env_init = Env.extend ("call/cc", l, Env.empty)
    val sto_init = s2
  end

  (* ... *)
  | eval (Syn.LAM (is, t), r, k, s, pg) =
    let val (l, s') = Sto.new (s, Sem.UNSPECIFIED)
      in k (Sem.PROCEDURE (fn (vs, k, s, pg)
        => if List.length vs = List.length is
          then let val (ls, s') = Sto.news (s, vs)
              in eval (t, Env.extends (is, ls, r), k, s', pg)
              end
          else Sem.STUCK "arity mismatch",
            1),
            s', pg)
      end
  (* ... *)
  and apply (Sem.PROCEDURE (f, _), vs, k, s, pg) =
    f (vs, k, s, pg)
  | apply (_ , vs, k, s, pg) =
    Sem.STUCK "attempt to apply a non-procedure"
  (* ... *)
end

```

---

**Fig. 13.** Closure-unconverted version of Figs. 10, 11, and 12 (only the modified parts)

## 11 Analysis

### 11.1 From Abstract Machine to Denotational Semantics

The evaluation function of Fig. 13 differs from the one in Clinger's denotational semantics (adapted to thread a stream of permutations) in two ways:

- the continuation domains are not the same: the operational continuations are passed an environment whereas the denotational ones are not;
- the operational treatment of sub-terms in an application differs from the denotational one: in the operational one, the sub-terms are not only permuted but the result of this permutation is reversed, so that the resulting values can be iteratively accumulated in the ‘`evlis`’ function.

## 11.2 From Denotational Semantics to Abstract Machine

Conversely, defunctionalizing Clinger’s denotational semantics (adapted to thread a stream of permutations) yields a big-step abstract machine that also differs from the one presented at PLDI’98: it is a traditional eval/continue abstract machine where the ‘`continu`’ transition function is not passed any environment. In particular, the permuted sub-terms are not reversed prior to be evaluated and the corresponding list of values is not accumulated.

## 11.3 Related Work

Earlier on, Biernacki and the author carried out the same experiment for Propositional Prolog with cut [10]. Comparing de Bruin and de Vink’s operational and denotational semantics [25], we found mismatches that are similar to the ones reported in this section.

For the rest, the literature is rich with connections and derivations between semantic artifacts. To the best of our knowledge, none are as simple and as effective as the ones pioneered by Reynolds and used here.

# 12 Conclusion and Perspectives

We have presented new semantic specifications for Core Scheme as specified by Clinger in his PLDI’98 article. These semantic specifications are compatible and inter-derived. It is our analysis that structurally, they differ from similar semantics that have independently been published.

There are two next logical steps to the present work, an analytical one and a constructive one:

- Redo this experiment on a larger scale with the other semantic specifications of Scheme. We will then be in position to compare all the small-step semantics, all the abstract machines, and all the big-step semantics of Scheme with respect to each other, and verify whether Scheme is uniformly and uniquely specified.
- Specify Scheme in part or in toto with inter-derivable specifications so that the compatibility of these specifications is a corollary of the correctness of the derivations. This corollary would let us flesh out, for example, Gasbichler, Knauel, and Sperber’s conjecture of equivalence for their operational and denotational semantics of Scheme with multiple threads [28, Section 6.4].

The author does not have any opinion about the particular goodness of one or another semantic specification of Scheme. He however feels strongly that Scheme's semantic artifacts should be compatible with each other.

*Acknowledgments.* The author is grateful to the anonymous reviewers of the 2008 Workshop on Scheme and Functional Programming and of the Mosses Festschrift for their comments, and to Will Clinger for extra explanations about his use of an infinite set of rules generated by a rule schema that is parameterized by a permutation and its inverse. Special thanks to Kevin Millikin and Ian Zerny for timely feedback on the workshop version and to Małgorzata Biernacka for joining forces in a 'Part II' companion article about reduction semantics and abstract machines [8].

This work is partly supported by the Danish Natural Science Research Council, Grant no. 21-03-0545. It is dedicated to Peter Mosses for his 60th birthday, in celebration for his successful career, which included 25 years in Aarhus.

## References

1. Ager, M.S.: Partial Evaluation of String Matchers & Constructions of Abstract Machines. PhD thesis, BRICS PhD School, Department of Computer Science, Aarhus University, Aarhus, Denmark (January 2006)
2. Ager, M.S., Biernacki, D., Danvy, O., Midtgaard, J.: A functional correspondence between evaluators and abstract machines. In: Miller, D. (ed.) Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP 2003), Uppsala, Sweden, August 2003, pp. 8–19. ACM Press, New York (2003)
3. Ager, M.S., Danvy, O., Midtgaard, J.: A functional correspondence between call-by-need evaluators and lazy abstract machines. Information Processing Letters 90(5), 223–232 (2004); Extended version available as the research report BRICS RS-04-3
4. Ager, M.S., Danvy, O., Midtgaard, J.: A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. Theoretical Computer Science 342(1), 149–172 (2005)
5. Biernacka, M.: A Derivational Approach to the Operational Semantics of Functional Languages. PhD thesis, BRICS PhD School, Department of Computer Science, Aarhus University, Aarhus, Denmark (January 2006)
6. Biernacka, M., Biernacki, D., Danvy, O.: An operational foundation for delimited continuations in the CPS hierarchy. Logical Methods in Computer Science 1(2:5), 1–39 (2005); A preliminary version was presented at the Fourth ACM SIGPLAN Workshop on Continuations (CW 2004)
7. Biernacka, M., Danvy, O.: A syntactic correspondence between context-sensitive calculi and abstract machines. Theoretical Computer Science 375(1-3), 76–108 (2007); Extended version available as the research report BRICS RS-06-18
8. Biernacka, M., Danvy, O.: Towards Compatible and Interderivable Semantic Specifications for the Scheme Programming Language, Part II: Reduction Semantics and Abstract Machines. In: Palsberg, J. (ed.) Mosses Festschrift. LNCS, vol. 5700, pp. 186–206. Springer, Heidelberg (2009)
9. Biernacki, D.: The Theory and Practice of Programming Languages with Delimited Continuations. PhD thesis, BRICS PhD School, Department of Computer Science, Aarhus University, Aarhus, Denmark (December 2005)

10. Biernacki, D., Danvy, O.: From interpreter to logic engine by defunctionalization. In: Bruynooghe, M. (ed.) LOPSTR 2003. LNCS, vol. 3018, pp. 143–159. Springer, Heidelberg (2003)
11. Clinger, W., Friedman, D.P., Wand, M.: A scheme for a higher-level semantic algebra. In: Reynolds, J., Nivat, M. (eds.) Algebraic Methods in Semantics, pp. 237–250. Cambridge University Press, Cambridge (1985)
12. Clinger, W., Hartheimer, A.H., Ost, E.M.: Implementation strategies for first-class continuations. Higher-Order and Symbolic Computation 12(1), 7–45 (1999); A preliminary version was presented at the 1988 ACM Conference on Lisp and Functional Programming
13. Clinger, W., Rees, J. (eds.): Revised<sup>4</sup> report on the algorithmic language Scheme. LISP Pointers IV(3), 1–55 (1991)
14. Clinger, W.D.: Proper tail recursion and space efficiency. In: Cooper, K.D. (ed.) Proceedings of the ACM SIGPLAN 1998 Conference on Programming Languages Design and Implementation, Montréal, Canada, pp. 174–185. ACM Press, New York (1998)
15. Danvy, O.: Formalizing implementation strategies for first-class continuations. In: Smolka, G. (ed.) ESOP 2000. LNCS, vol. 1782, pp. 88–103. Springer, Heidelberg (2000)
16. Danvy, O.: A rational deconstruction of Landin’s SECD machine. In: Grelck, C., Huch, F., Michaelson, G.J., Trinder, P. (eds.) IFL 2004. LNCS, vol. 3474, pp. 52–71. Springer, Heidelberg (2004); Recipient of the 2004 Peter Landin prize. Extended version available as the research report BRICS RS-03-33
17. Danvy, O.: An Analytical Approach to Program as Data Objects. DSc thesis, Department of Computer Science, Aarhus University, Aarhus, Denmark (October 2006)
18. Danvy, O., Filinski, A.: Representing control, a study of the CPS transformation. Mathematical Structures in Computer Science 2(4), 361–391 (1992)
19. Danvy, O., Millikin, K.: Refunctionalization at work. Science of Computer Programming 74(8), 534–549 (2009)
20. Danvy, O., Millikin, K.: On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion. Information Processing Letters 106(3), 100–109 (2008)
21. Danvy, O., Millikin, K.: Refunctionalization at work. Science of Computer Programming 74(8), 534–549 (2009); Extended version available as the research report BRICS SCP09
22. Danvy, O., Nielsen, L.R.: Defunctionalization at work. In: Søndergaard, H. (ed.) Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2001), Firenze, Italy, September 2001, pp. 162–174. ACM Press, New York (2001); Extended version available as the research report BRICS RS-01-23
23. Danvy, O., Nielsen, L.R.: Refocusing in reduction semantics. Research Report BRICS RS-04-26, Department of Computer Science, Aarhus University, Aarhus, Denmark (November 2004); A preliminary version appeared in the informal proceedings of the Second International Workshop on Rule-Based Programming (RULE 2001), Electronic Notes in Theoretical Computer Science, vol. 59.4 (2001)
24. Danvy, O., Yang, Z.: An operational investigation of the CPS hierarchy. In: Swierstra, S.D. (ed.) ESOP 1999. LNCS, vol. 1576, pp. 224–242. Springer, Heidelberg (1999)

25. de Bruin, A., de Vink, E.P.: Continuation semantics for Prolog with cut. In: Díaz, J., Orejas, F. (eds.) TAPSOFT 1989. LNCS, vol. 351, pp. 178–192. Springer, Heidelberg (1989)
26. Felleisen, M., Flatt, M.: Programming languages and lambda calculi (1989–2001) (unpublished lecture notes), <http://www.ccs.neu.edu/home/matthias/3810-w02/readings.html> (last accessed in April 2008)
27. Felleisen, M., Friedman, D.P.: Control operators, the SECD machine, and the  $\lambda$ -calculus. In: Wirsing, M. (ed.) Formal Description of Programming Concepts III, pp. 193–217. Elsevier Science Publishers B.V (North-Holland), Amsterdam (1986)
28. Gasbichler, M., Knauel, E., Sperber, M.: How to add threads to a sequential language without getting tangled up. In: Flatt, M. (ed.) Proceedings of the Fourth ACM SIGPLAN Workshop on Scheme and Functional Programming, Technical report UUCS-03-023, School of Computing, University of Utah, Boston, Massachusetts, November 2003, pp. 30–47 (2003)
29. Griffin, T.G.: A formulae-as-types notion of control. In: Hudak, P. (ed.) Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, January 1990, pp. 47–58. ACM Press, New York (1990)
30. Harper, R., Duba, B.F., MacQueen, D.: Typing first-class continuations in ML. Journal of Functional Programming 3(4), 465–484 (1993); A preliminary version was presented at the Eighteenth Annual ACM Symposium on Principles of Programming Languages (POPL 1991)
31. Haynes, C.T., Friedman, D.P., Wand, M.: Continuations and coroutines. In: Steele Jr., G.L. (ed.) Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming, Austin, Texas, August 1984, pp. 293–298. ACM Press, New York (1984)
32. Hieb, R., Dybvig, R.K., Bruggeman, C.: Representing control in the presence of first-class continuations. In: Lang, B. (ed.) Proceedings of the ACM SIGPLAN 1990 Conference on Programming Languages Design and Implementation, White Plains, June 1990. SIGPLAN Notices, vol. 25(6), pp. 66–77. ACM Press, New York (1990)
33. Johannsen, J.: An investigation of Abadi and Cardelli's untyped calculus of objects. Master's thesis, Department of Computer Science, Aarhus University, Aarhus, Denmark, BRICS research report RS-08-6 (June 2008)
34. Kelsey, R., Clinger, W., Rees, J.: Revised<sup>5</sup> report on the algorithmic language Scheme. Higher-Order and Symbolic Computation 11(1), 7–105 (1998)
35. Krivine, J.-L.: A call-by-name lambda-calculus machine. Higher-Order and Symbolic Computation 20(3), 199–207 (2007)
36. Landin, P.J.: The mechanical evaluation of expressions. The Computer Journal 6(4), 308–320 (1964)
37. Marlow, S., Peyton Jones, S.L.: Making a fast curry: push/enter vs. eval/apply for higher-order languages. In: Fisher, K. (ed.) Proceedings of the 2004 ACM SIGPLAN International Conference on Functional Programming (ICFP 2004), September 2004. SIGPLAN Notices, vol. 39(9), pp. 4–15. ACM Press, New York (2004)
38. Matthews, J., Findler, R.B.: An operational semantics for Scheme. Journal of Functional Programming 18(1), 47–86 (2008)
39. Midtgaard, J.: Transformation, Analysis, and Interpretation of Higher-Order Procedural Programs. PhD thesis, BRICS PhD School, Aarhus University, Aarhus, Denmark (June 2007)

40. Millikin, K.: A Structured Approach to the Transformation, Normalization and Execution of Computer Programs. PhD thesis, BRICS PhD School, Aarhus University, Aarhus, Denmark (May 2007)
41. Milner, R., Tofte, M., Harper, R., MacQueen, D.: The Definition of Standard ML (Revised). MIT Press, Cambridge (1997)
42. Munk, J.: A study of syntactic and semantic artifacts and its application to lambda definability, strong normalization, and weak normalization in the presence of state. Master's thesis, Department of Computer Science, Aarhus University, Aarhus, Denmark, BRICS research report RS-08-3 (May 2007)
43. Nielsen, L.R.: A study of defunctionalization and continuation-passing style. PhD thesis, BRICS PhD School, Department of Computer Science, Aarhus University, Aarhus, Denmark (July 2001) BRICS DS-01-7
44. Ohori, A., Sasano, I.: Lightweight fusion by fixed point promotion. In: Felleisen, M. (ed.) Proceedings of the Thirty-Fourth Annual ACM Symposium on Principles of Programming Languages, January 2007. SIGPLAN Notices, vol. 42(1), pp. 143–154. ACM Press, New York (2007)
45. Plotkin, G.D.: Call-by-name, call-by-value and the  $\lambda$ -calculus. Theoretical Computer Science 1, 125–159 (1975)
46. Ramsdell, J.D.: An operational semantics for Scheme. LISP Pointers V(2), 6–10 (1992)
47. Rees, J., Clinger, W. (eds.): Revised<sup>3</sup> report on the algorithmic language Scheme. SIGPLAN Notices 21(12), 37–79 (1986)
48. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. In: Proceedings of 25th ACM National Conference, Boston, Massachusetts, pp. 717–740 (1972); Reprinted in Higher-Order and Symbolic Computation 11(4), 363–397 (1998), with a foreword [50]
49. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. Higher-Order and Symbolic Computation 11(4), 363–397 (1998); Reprinted from the proceedings of the 25th ACM National Conference (1972), with a foreword [50]
50. Reynolds, J.C.: Definitional interpreters revisited. Higher-Order and Symbolic Computation 11(4), 355–361 (1998)
51. Sperber, M., Kent Dybvig, R., Flatt, M., van Straaten, A. (eds.): Revised<sup>6</sup> report on the algorithmic language Scheme (September 2007), <http://www.r6rs.org/>
52. Steele Jr., G.L.: Rabbit: A compiler for Scheme. Master's thesis, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, Technical report AI-TR-474 (May 1978)
53. Steele Jr., G.L., Sussman, G.J.: The revised report on Scheme, a dialect of Lisp. AI Memo 452, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts (January 1978)
54. Sussman, G.J., Steele Jr., G.L.: Scheme: An interpreter for extended lambda calculus. AI Memo 349, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts (December 1975); Reprinted in Higher-Order and Symbolic Computation 11(4), 405–439 (1998), with a foreword [55]
55. Sussman, G.J., Steele Jr., G.L.: The first report on Scheme revisited. Higher-Order and Symbolic Computation 11(4), 399–404 (1998)
56. Wadler, P.: The essence of functional programming (invited talk). In: Appel, A.W. (ed.) Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, January 1992, pp. 1–14. ACM Press, New York (1992)

# Towards Compatible and Interderivable Semantic Specifications for the Scheme Programming Language, Part II: Reduction Semantics and Abstract Machines

Małgorzata Biernacka<sup>1</sup> and Olivier Danvy<sup>2</sup>

<sup>1</sup> Institute of Computer Science, University of Wrocław  
ul. Joliot-Curie 15, PL-50-383 Wrocław, Poland  
[mabi@ii.uni.wroc.pl](mailto:mabi@ii.uni.wroc.pl)

<sup>2</sup> Department of Computer Science, Aarhus University  
Aabogade 34, DK-8200 Aarhus N, Denmark  
[danvy@cs.au.dk](mailto:danvy@cs.au.dk)

**Abstract.** We present a context-sensitive reduction semantics for a lambda-calculus with explicit substitutions and we show that the functional implementation of this small-step semantics mechanically corresponds to that of the abstract machine for Core Scheme presented by Clinger at PLDI'98, including first-class continuations. Starting from this reduction semantics, (1) we refocus it into a small-step abstract machine; (2) we fuse the transition function of this abstract machine with its driver loop, obtaining a big-step abstract machine which is staged; (3) we compress its corridor transitions, obtaining an eval/continue abstract machine; and (4) we unfold its ground closures, which yields an abstract machine that essentially coincides with Clinger's machine. This lambda-calculus with explicit substitutions therefore aptly accounts for Core Scheme, including Clinger's permutations and unpermutations.

## 1 Introduction

*Motivation:* Our motivation is the same as that of the second author in the companion paper “Towards Compatible and Interderivable Semantic Specifications for the Scheme Programming Language, Part I: Denotational Semantics, Natural Semantics, and Abstract Machines” [10]. We wish for semantic specifications that are mechanically interderivable, so that their compatibility is a corollary of the correctness of the derivations.

*This work:* We build on our previous work on the syntactic correspondence between context-sensitive reduction semantics and abstract machines for a  $\lambda$ -calculus with explicit substitutions [3, 4]. Let us review each of these concepts in turn:

**Abstract machines:** An abstract machine is a state-transition system modeling the execution of programs. Typical abstract machines for lambda calculi treat substitution as a meta-operation and include it directly in the

transitions of the machine. This approach is often used to faithfully model evaluation based on term rewriting. Alternatively, since Landin’s SECD machine [12, 15], substitution is explicitly implemented in abstract machines using environments, for efficiency. The two approaches are used interchangeably in the literature (even for the same language), depending on the context of use, but their equivalence is rarely treated formally.

**A  $\lambda$ -calculus with explicit substitutions:** Since Plotkin’s foundational work on  $\lambda$ -calculi and abstract machines [19], it has become a tradition to directly relate the result of abstract machines with the result of weak-head normalization, regardless of whether the abstract machines treat substitution implicitly as a meta-operation or explicitly with an environment. As an off-shoot of his doctoral thesis [6, 7], Curien proposed a ‘calculus of closures,’ the  $\lambda\rho$ -calculus, that would, on the one hand, be faithful to the  $\lambda$ -calculus, and on the other hand, reflect more accurately the computational reality of abstract machines by delaying substitutions into environments. In so doing he gave birth to calculi of explicit substitutions [1], which promptly became a domain of research on their own.

In our thesis work [2, 8], we revisited the  $\lambda\rho$ -calculus and proposed a minimal extension for it, the  $\lambda\widehat{\rho}$ -calculus, that is closed under one-step reduction. We then systematically applied Danvy and Nielsen’s refocusing technique [13] on several reduction semantics and obtained a variety of known and new abstract machines with environments, including the Krivine machine for call by name and the CEK machine for call by value [3].

**Context-sensitive reduction semantics:** In his thesis work [14], Felleisen introduced a continuation-semantics analogue of structural operational semantics, reduction semantics: a small-step operational semantics with an explicit representation of the reduction context. As Strachey and Wadsworth originally did with continuation semantics [21], he then took advantage of this explicit representation of the rest of the reduction to make contraction rules context sensitive, and operate not just on a potential redex,<sup>1</sup> but also on its context, thereby providing the first small-step semantic account of control operators.

In our thesis work [2, 8], we considered context-sensitive contraction rules for  $\lambda\widehat{\rho}$ -calculi. We then systematically applied the refocusing technique on several context-sensitive reduction semantics and obtained a variety of known and new abstract machines with environments [4].

In this article, we present a variant of the  $\lambda\widehat{\rho}$ -calculus that, through refocusing, essentially corresponds to Clinger’s abstract machine for Core Scheme as presented at PLDI’98 [5]. Curien’s original point therefore applies and reductions in this calculus reflect the execution of Scheme programs accurately. We therefore put the  $\lambda\widehat{\rho}$ -calculus forward as an apt calculus for Core Scheme.

*Prerequisites and domain of discourse:* Though one could of course use PLT Redex [17], we use a pure subset of Standard ML as a metalanguage here, for

---

<sup>1</sup> A potential redex either is an actual one or is stuck.

consistency with the companion paper. We otherwise expect some familiarity with programming a reduction semantics and with Clinger’s PLDI’98 article [5]. For the rest, we have aimed for a stand-alone presentation but the reader might wish to consult our earlier work [3, 4] or first flip through the pages of the second author’s lecture notes for warm-up examples [9].

### *Terminology*

**Notion of contraction:** To alleviate the overloading of the term ‘reduction’ (as in, e.g., “reduction semantics,” “notion of reduction,” “reduction strategy,” and “reduction step”), we refer to Barendregt’s ‘notion of reduction’ as ‘notion of contraction.’ A notion of contraction is therefore the definition of a partial contraction function mapping a potential redex to a contractum.

**Eval/continue abstract machine:** As pointed out in the companion paper, an ‘eval/apply’ abstract machine [16] would be more accurately called ‘eval/-continue’ since the apply transition function, together with the data type of contexts, often form the defunctionalized representation of a continuation. We therefore use this term here.

*Overview:* We first present the signatures of the store, the environment, and the permutations (Section 2), and then the syntax (Section 3) and the reduction semantics (Sections 4 and 5) of a  $\lambda$ -calculus with explicit substitutions for Core Scheme. The resulting evaluation function is reduction-based in that it is defined as the iteration of a one-step reduction function that enumerates all the intermediate closures in the reduction sequence. We make it reduction-free by deforesting all these intermediate closures in the course of evaluation, using Danvy and Nielsen’s refocusing technique (Section 6). We successively present an eval/continue abstract machine over closures that embodies the chosen reduction strategy (Section 6.1), and then an eval/continue abstract machine over terms and environments (Section 6.2). We then analyze this machine (Section 7) before concluding (Section 8).

## 2 Domain of Discourse

In the interest of brevity and abstractness, and as in the companion paper, we only present ML signatures for the store (Section 2.1), the environment (Section 2.2) and the permutations (Section 2.3).

### 2.1 Store

A store is a mapping from locations to storable values. We specify it as a polymorphic abstract data type with the usual algebraic operators to allocate fresh locations and initialize them with given expressible values, dereference a given location in a given store, and update a given store at a given location with a given expressible value.

```

signature STO = sig
  type 'a sto
  type loc

  val empty : 'a sto

  val new : 'a sto * 'a      -> loc      * 'a sto
  val news : 'a sto * 'a list -> loc list * 'a sto

  val fetch : loc *      'a sto -> 'a option
  val update : loc * 'a * 'a sto -> 'a sto option
end

structure Sto :> STO = struct
  (* deliberately omitted *)
end

```

In his definition of Core Scheme [5, Fig. 4], Clinger make storables values and expressible values coincide [20].

## 2.2 Environment

An environment is a mapping from identifiers to denotable values. We specify it as a polymorphic abstract data type with the usual algebraic operators to extend a given environment with new bindings and to look up identifiers in a given environment, for a given type of identifiers.

```

type ide = string

signature ENV = sig
  type 'a env

  val empty : 'a env
  val emptyp : 'a env -> bool

  val extend : ide      * 'a      * 'a env -> 'a env
  val extends : ide list * 'a list * 'a env -> 'a env

  val lookup : ide * 'a env -> 'a option
end

structure Env :> ENV = struct
  (* deliberately omitted *)
end

```

In the definition of Scheme, the denotable values are store locations.

## 2.3 Permutations

The semantics of Scheme deliberately does not specify the order in which the subterms of an application are evaluated. This underspecification (also present in C for assignments) is meant to encourage programmers to explicitly sequence their side effects.

To this end, in his abstract machine, Clinger non-deterministically uses a pair of permutation functions: one over the subterms in an application, and the inverse one over the resulting values. We implement this non-determinism by threading a stream of pairs of permutations and unpermutations along with the store. We materialize this stream with the following polymorphic abstract data type.

```

signature PERM = sig
  type 'a perm
  type ('v, 'c) permgen

  val init : ('v, 'c) permgen
  val new : ('v, 'c) permgen -> ('v perm * 'c perm) * ('v, 'c) permgen
end

structure Perm :> PERM = struct
  (* deliberately omitted *)
end

```

### 3 Syntax

The following module implements the internal syntax of Core Scheme [5, Fig. 1], for a given type of identifiers.

```

structure Syn = struct
  datatype quotation = QBOOL of bool
    | QNUMB of int
    | QSYMB of ide
    | QPAIR of Sto.loc * Sto.loc
    (* / QVECT of ... *)
    (* / ... *)

  datatype term = QUOTE of quotation
    | VAR of ide
    | LAM of ide list * term
    | APP of term * term list
    | COND of term * term * term
    | SET of ide * term
end

```

Terms include all the constructs of Core Scheme considered by Clinger: quoted values, identifiers, lambda abstractions, applications, conditional expressions and assignments. Primitive operators such as call/cc are declared in the initial environment.

### 4 Semantics

We consider a language of closures built on top of terms. Fig. 1 displays the syntactic categories of closures (the data type `closure`) and of contexts (the data type `cont`) as well as the notion of environment, value, store, permutation generator, and answer. Let us review each of these in turn.

---

```

structure Sem = struct
  type env = Sto.loc Env.env

  datatype primop = CWCC (* / ... *)

  datatype clo =
    CLO_GND of Syn.term * env
  | CLO_QUOTE of Syn.quotation
  | CLO_LAM of ide list * Syn.term * env * Sto.loc
  | CLO_APP of clo * clo list * value list * value Perm.perm
  | CLO_CALL of value * value list
  | CLO_COND of clo * clo * clo
  | CLO_SET of ide * env * clo
  | CLO_UNSPECIFIED
  | CLO_UNDEFINED
  | CLO_PRIMOP of primop * Sto.loc
  | CLO_CONT of cont * Sto.loc
and cont =
  HALT
  | SELECT of clo * clo * cont
  | ASSIGN of ide * env * cont
  | PUSH of clo list * value list * value Perm.perm * cont
  | CALL of value list * cont
withtype value = clo

  type sto = value Sto.sto

  type perms = (value, clo) Perm.permgen

  datatype answer = VALUE of value * sto * perms
    | STUCK of string

  local val (l_primop, s1) = Sto.new (Sto.empty, CLO_UNSPECIFIED)
    val (l_cwcc, s2) = Sto.new (s1, CLO_PRIMOP (CWCC, l_primop))
  in val env_init = Env.extend ("call/cc", l_cwcc, Env.empty)
    val sto_init = s2
  end
end

```

---

**Fig. 1.** Lambda-calculus with explicit substitutions for Core Scheme

## 4.1 The Environment

As described in Section 2.2, the environment maps identifiers to denotable values, and denotable values are store locations.

## 4.2 Closures

As initiated by Landin [15] and continued by Curien [6, 7], a ground closure is a term paired with a syntactic representation of its environment (this pairing is done with the constructor CLO\_GND). In a calculus of closures, small-step evaluation is defined (e.g., by a set of rewriting rules) over closures rather than over terms. Ground closures, however, are usually not expressive enough to account for one-step reductions, though they may suffice for big-step evaluation. Indeed, one-step reduction can require the internal structure of a closure to be changed in such a way that it no longer conforms to the form “(term, environment).” The data type of closures therefore contains additional constructors to represent intermediate results of one-step reductions for all the language constructs.

- The `CLO_GND` constructor is used for ground closures.
- The `CLO_QUOTE` constructor accounts for Scheme’s quotations.
- The `CLO_LAM` constructor accounts for user-defined procedures, and pairs lambda-abstractions (list of formal parameters and body) together with the environment of their definition.
- The `CLO_APP` constructor accounts for applications whose subcomponents are not completely reduced yet. The closure and list of closures still need to be reduced, and the list of values holds what has already been reduced. The value permutation will be used to unpermute the complete list of values and yield the `CLO_CALL` construction.
- The `CLO_CALL` constructor accounts for applications whose subcomponents are completely reduced.
- The `CLO_COND` constructor accounts for conditional closures.
- The `CLO_SET` constructor accounts for assignments.
- The `CLO_UNSPECIFIED` constructor accounts for the unspecified value yielded, e.g., by reducing an assignment.
- The `CLO_UNSPECIFIED` constructor accounts for the undefined value.
- The `CLO_PRIMOP` constructor accounts for predefined procedures (i.e., primitive operators) such as “call-with-current-continuation” (commonly abbreviated “call/cc”), that captures the current context.
- The `CLO_CONT` constructor accounts for escape procedures, i.e., first-class continuations as yielded by call/cc. It holds a captured context (see Section 4.5).

**Note:** In Scheme, procedures of course cannot be compared for mathematical equality, but they can be compared for representational identity. So for example, `(equal? (lambda (x) x) (lambda (x) x))` evaluates to #*f* but `(let ([identity (lambda (x) x)]) (equal? identity identity))` evaluates to #*t*. For better or for worse,<sup>2</sup> a unique location is associated to every applicable object, be it a user-defined procedure, a predefined procedure, or an escape procedure. This is the reason why the closure constructors `CLO_LAM`, `CLO_PRIMOP`, and `CLO_CONT` feature a store location.

### 4.3 Primitive Operators

The data type `primop` groups all the predefined procedures. Here, we only consider one, `call/cc`.

### 4.4 Values

(Expressible) values are closures that cannot be decomposed into a potential redex and its reduction context, namely quotations, lambda-abstractions, the unspecified value, the undefined value, primitive operators, and first-class continuations:

---

<sup>2</sup> Will Clinger publically refers to this particular design as a “bug” in the semantics of Scheme.

```
(* valuep : Sem.clo -> bool *)
fun valuep (CLO_QUOTE _)      = true
| valuep (CLO_LAM _)          = true
| valuep CLO_UNSPECIFIED     = true
| valuep CLO_UNDEFINED       = true
| valuep (CLO_PRIMOP _)      = true
| valuep (CLO_CONT _)         = true
| valuep _                     = false
```

## 4.5 Contexts

The data type faithfully reflects Clinger's grammar of contexts [5, Fig. 4]:

- The HALT constructor accounts for the empty context.
- The SELECT constructor accounts for the context of the test part of a conditional expression.
- The ASSIGN constructor accounts for the context of the subcomponent in an assignment.
- The PUSH constructor accounts for the context of a subcomponent in a permuted application.
- The CALL constructor accounts for the context of a value in position of function in an unpermuted application of values.

## 4.6 The Store

As described in Section 2.1, the store maps locations to storable values, and storable values are either NONE for the undefined value or SOME  $v$  for the expressible value  $v$ .

## 4.7 The Permutation Generator

As described in Section 2.3, we thread a stream of pairs of permutations (of closures) and unpermutations (of values) along with the store.

## 4.8 Answers

Any non-diverging reduction sequence starting from a closure either leads to a value closure or becomes stuck. The result of a non-diverging evaluation is an answer, i.e., either an expressible value (as defined in Section 4.4) together with a store and a permutation generator, or an error message.

## 4.9 The Initial Store

The initial store holds the values of primitive operators such as call/cc.

## 4.10 The Initial Environment

The initial environment declares primitive operators such as call/cc.

---

```

structure Redexes = struct
  datatype potred =
    LOOKUP of ide * Sem.env
    UNPERMUTE of Sem.value * Sem.value list * Sem.value Perm.perm
    BETA of Sem.value * Sem.value list
    UPDATE of ide * Sem.env * Sem.value
    COND of Sem.value * Sem.clo * Sem.clo
    PROC of ide list * Syn.term * Sem.env
    PROP_APP of Syn.term * Syn.term list * Sem.env
    PROP_COND of Syn.term * Syn.term * Syn.term * Sem.env
    PROP_SET of ide * Syn.term * Sem.env

  datatype contractum =
    STUCK of string
    | NEXT of Sem.clo * Sem.cont * Sem.sto * Sem.perms

  (* contract : potred * Sem.cont * Sem.sto * Sem.perms -> contractum *)
  fun contract ... =
    ...
end

```

---

**Fig. 2.** Notion of contraction for Core Scheme (part 1/2)

## 5 A Reduction Semantics for Core Scheme

A reduction semantics is a small-step operational semantics with an explicit representation of the reduction context. It consists of a grammar of terms (here, the grammar of closures from Fig. 1), a notion of contraction specifying the basic computation steps, and a reduction strategy embodied by a grammar of reduction contexts (see Fig. 1). In this section, we present a reduction semantics for the calculus of closures introduced in Section 4.

### 5.1 Potential Redexes and Contraction

The notion of contraction is defined with two data types—one for potential redexes and one for the result of contraction (Figure 2)—and with a contraction function (Figure 3). Let us review each of these potential redexes and how they are contracted:

- LOOKUP – fetching the value of an identifier from the store (via its location in the environment); it succeeds only if the location corresponding to the identifier is defined in the store – otherwise reduction is stuck;
- UNPERMUTE – performing the unpermutation of a sequence of values before applying BETA-contraction;
- BETA – either performing the usual  $\beta$ -reduction for n-ary functions (when the operand is a user-defined procedure), or capturing the current context (when the operand is call/cc), or replacing the current context by a captured context (when the operand is an escape procedure);
- UPDATE – updating the value of an identifier in the store and returning the “unspecified” closure;

---

```

fun contract (LOOKUP (i, r), rc, s, pg) =
  (case Env.lookup (i, r)
   of (SOME l)
     => (case Sto.fetch (l, s)
          of (SOME v)
            => (case v
                  of Sem.CLO_UNDEFINED
                    => STUCK "undefined value"
                  | _
                    => NEXT (v, rc, s, pg))
          | NONE
            => STUCK "attempt to read an invalid location")
   | NONE
     => STUCK "attempt to reference an undeclared variable")
| contract (UNPERMUTE (v, vs, pi), rc, s, pg) =
  NEXT (Sem.CLO_CALL (pi (v, vs)), rc, s, pg)
| contract (BETA (Sem.CLO_LAM (is, t, r, l), vs), rc, s, pg) =
  if List.length is = List.length vs
  then let val (ls, s') = Sto.news (s, vs)
    in NEXT (Sem.CLO_GND (t, Env.extends (is, ls, r)), rc, s', pg)
    end
  else STUCK "arity mismatch"
| contract (BETA (Sem.CLO_PRIMOP (Sem.CWCC, _), vs), rc, s, pg) =
  if l = List.length vs
  then let val (l, s') = Sto.new (s, Sem.CLO_UNSPECIFIED)
    in NEXT (Sem.CLO_CALL (hd vs, [Sem.CLO_CONT (rc, l)]), rc, s', pg)
    end
  else STUCK "arity mismatch"
| contract (BETA (Sem.CLO_CONT (rc', l), vs), rc, s, pg) =
  if l = List.length vs
  then NEXT (hd vs, rc', s, pg)
  else STUCK "arity mismatch"
| contract (BETA (_, vs), rc, s, pg) =
  STUCK "attempt to apply a non-procedure"
| contract (UPDATE (i, r, v), rc, s, pg) =
  (case Env.lookup (i, r)
   of (SOME l)
     => (case Sto.update (l, v, s)
          of (SOME s')
            => NEXT (Sem.CLO_UNSPECIFIED, rc, s', pg)
          | NONE
            => STUCK "attempt to write an invalid location")
   | NONE
     => STUCK "attempt to assign an undeclared variable")
| contract (COND (Sem.CLO_QUOTE (Syn.QBOOL false), c1, c2), rc, s, pg) =
  NEXT (c2, rc, s, pg)
| contract (COND (_, c1, c2), rc, s, pg) =
  NEXT (c1, rc, s, pg)
| contract (PROC (is, t, r), rc, s, pg) =
  let val (l, s') = Sto.new (s, Sem.CLO_UNSPECIFIED)
  in NEXT (Sem.CLO_LAM (is, t, r, l), rc, s', pg) end
| contract (PROP_APP (t, ts, r), rc, s, pg) =
  let val ((pi, rev_pi_inv), pg') = Perm.new pg
    val (c, cs) = rev_pi_inv (Sem.CLO_GND (t, r),
                               map (fn t => Sem.CLO_GND (t, r)) ts)
  in NEXT (Sem.CLO_APP (c, cs, nil, pi), rc, s, pg') end
| contract (PROP_COND (t0, t1, t2, r), rc, s, pg) =
  NEXT (Sem.CLO_COND (Sem.CLO_GND (t0, r),
                      Sem.CLO_GND (t1, r),
                      Sem.CLO_GND (t2, r))), rc, s, pg)
| contract (PROP_SET (i, t, r), rc, s, pg) =
  NEXT (Sem.CLO_SET (i, r, Sem.CLO_GND (t, r))), rc, s, pg)

```

---

**Fig. 3.** Notion of contraction for Core Scheme (part 2/2)

- COND – selecting one of the branches of a conditional expression, based on the value of its test;
- PROC – allocating a fresh location in the store (with an unspecified value) for a source lambda abstraction and converting this abstraction to CLO\_LAM;
- PROP\_APP, PROP\_COND, and PROP\_SET – propagating the environment into all subterms of an application, a conditional expression, and an assignment, respectively. Beside environment propagation, in PROP\_APP all components of the application are permuted.

While most of the contractions account directly for the reductions in the language, the last three – the propagation contractions – are “administrative” reductions necessary to maintain the proper syntactic structure of closures after each reduction step. In addition, PROP\_APP includes a permutation of all components of an application before they are evaluated in turn.

The notion of contraction depends not only on closures but also on the reduction context that can be captured by call/cc, on the store, and on the permutation generator. Therefore, all three are supplied as arguments to the contract function.

## 5.2 Reduction Strategy

The reduction strategy is embodied in the grammar of reduction contexts defined by the data type cont in Fig. 1.

*Recomposition:* The function recompose reconstructs a closure given a reduction context and a (sub)closure. Its definition is displayed in Fig. 4.

*Decomposition:* The role of the decomposition function is to traverse a closure in a context according to the given reduction strategy and to locate the first redex to be contracted, if there is any. The decomposition function is total: it returns the closure if this closure is a value, and otherwise it returns a potential redex together with its reduction context. Its implementation is displayed in Fig. 5. In particular, decompose is called at the top level and its role is to call

```

structure Recomposition = struct
  (* recompose : Sem.cont * Sem.clo -> Sem.clo *)
  fun recompose (Sem.HALT, c) =
    c
  | recompose (Sem.SELECT (c1, c2, rc), c) =
    recompose (rc, Sem.CLO_COND (c, c1, c2))
  | recompose (Sem.ASSIGN (i, r, rc), c) =
    recompose (rc, Sem.CLO_SET (i, r, c))
  | recompose (Sem.PUSH (cs, vs, p, rc), c) =
    recompose (rc, Sem.CLO_APP (c, cs, vs, p))
  | recompose (Sem.CALL (vs, rc), c) =
    recompose (rc, Sem.CLO_CALL (c, vs))
end

```

**Fig. 4.** The recomposition function for Core Scheme

---

```

structure Decomposition = struct
  datatype decomposition = VAL of Sem.value
    | DEC of Redexes.potred * Sem.cont

  (* decompose_clo : Sem.clo * Sem.cont -> decomposition *)
  fun decompose_clo (Sem.CLO_GND (Syn.QUOTE q, r), rc) =
    decompose_cont (rc, Sem.CLO_QUOTE q)
  | decompose_clo (Sem.CLO_GND (Syn.VAR i, r), rc) =
    DEC (Redexes.LOOKUP (i, r), rc)
  | decompose_clo (Sem.CLO_GND (Syn.LAM (is, t), r), rc) =
    DEC (Redexes.PROC (is, t, r), rc)
  | decompose_clo (Sem.CLO_GND (Syn.APP (t, ts), r), rc) =
    DEC (Redexes.PROP_APP (t, ts, r), rc)
  | decompose_clo (Sem.CLO_GND (Syn.COND (t0, t1, t2), r), rc) =
    DEC (Redexes.PROP_COND (t0, t1, t2, r), rc)
  | decompose_clo (Sem.CLO_GND (Syn.SET (i, t), r), rc) =
    DEC (Redexes.PROP_SET (i, t, r), rc)
  | decompose_clo (v as Sem.CLO_QUOTE _, rc) =
    decompose_cont (rc, v)
  | decompose_clo (v as Sem.CLO_LAM _, rc) =
    decompose_cont (rc, v)
  | decompose_clo (Sem.CLO_APP (c, cs, vs, p), rc) =
    decompose_clo (c, Sem.PUSH (cs, vs, p, rc))
  | decompose_clo (Sem.CLO_CALL (v, vs), rc) =
    decompose_cont (Sem.CALL (vs, rc), v)
  | decompose_clo (Sem.CLO_COND (c0, c1, c2), rc) =
    decompose_clo (c0, Sem.SELECT (c1, c2, rc))
  | decompose_clo (Sem.CLO_SET (i, r, c1), rc) =
    decompose_clo (c1, Sem.ASSIGN (i, r, rc))
  | decompose_clo (v as Sem.CLO_UNSPECIFIED, rc) =
    decompose_cont (rc, v)
  | decompose_clo (v as Sem.CLO_UNDEFINED, rc) =
    decompose_cont (rc, v)
  | decompose_clo (v as Sem.CLO_PRIMOP (Sem.CWCC, _), rc) =
    decompose_cont (rc, v)
  | decompose_clo (v as Sem.CLO_CONT _, rc) =
    decompose_cont (rc, v)

  (* decompose_cont : Sem.cont * Sem.value -> decomposition *)
  and decompose_cont (Sem.HALT, v) =
    VAL v
  | decompose_cont (Sem.SELECT (c1, c2, rc), c) =
    DEC (Redexes.COND (c, c1, c2), rc)
  | decompose_cont (Sem.ASSIGN (i, r, rc), c) =
    DEC (Redexes.UPDATE (i, r, c), rc)
  | decompose_cont (Sem.PUSH (nil, vs, p, rc), v) =
    DEC (Redexes.UNPERMUTE (v, vs, p), rc)
  | decompose_cont (Sem.PUSH (c :: cs, vs, p, rc), v) =
    decompose_clo (c, Sem.PUSH (cs, v :: vs, p, rc))
  | decompose_cont (Sem.CALL (vs, rc), v) =
    DEC (Redexes.BETA (v, vs), rc)

  (* decompose : Sem.clo -> decomposition *)
  fun decompose c =
    decompose_clo (c, Sem.HALT)
end

```

---

**Fig. 5.** The decomposition function for Core Scheme

an auxiliary function, `decompose_clo`, with a closure to decompose and the empty context. In turn, `decompose_clo` traverses a closure and accumulates the current context until a potential redex or a value closure is found; in the latter case, `decompose_cont` is called in order to dispatch on the accumulated context for this given value.

The decomposition function can be expressed in a variety of ways. In Fig. 5, we have conveniently specified it as a big-step abstract machine with two transition functions, `decompose_clo` and `decompose_cont`.

### 5.3 One-Step Reduction

One-step reduction can now be defined with the following steps: (a) decomposing a non-value closure into a potential redex and a reduction context, (b) contracting the potential redex if it is an actual one, and (c) recomposing the contractum into the context.

```
(* reduce : Clo.clo * Clo.sto -> Clo.clo option *)
fun reduce (c, s, pg) =
  (case Decomposition.decompose c
   of (Decomposition.VAL v)
    => SOME (v, s, pg)
   | (Decomposition.DEC (pr, rc))
    => (case Redexes.contract (pr, rc, s, pg)
          of (Redexes.NEXT (c', rc', s', pg'))
           => SOME (Recomposition.recompose (rc', c'), s', pg')
          | (Redexes.STUCK msg)
           => NONE))
```

### 5.4 Reduction-Based Evaluation

Finally, we can define evaluation as the iteration of one-step reduction. We implement it as the iteration of (a) decomposition, (b) contraction, and (c) recomposition.

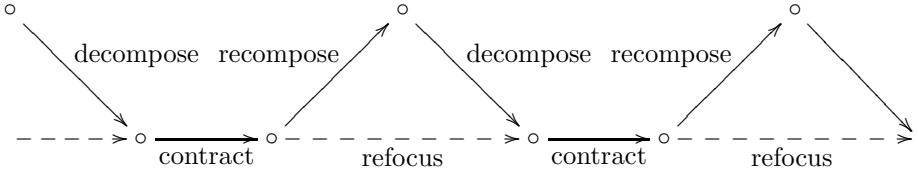
```
(* iterate : Decomposition.decomposition * Sem.sto -> Sem.answer *)
fun iterate (Decomposition.VAL v, s, pg)
  = Sem.VALUE (v, s, pg)
| iterate (Decomposition.DEC (pr, rc), s, pg)
  = (case Redexes.contract (pr, rc, s, pg)
       of Redexes.NEXT (c', rc', s', pg')
        => let val c = Recomposition.recompose (rc', c')
            val d = Decomposition.decompose c
            in iterate (d, s', pg')
            end
   | Redexes.STUCK msg
    => Sem.STUCK msg)

(* evaluate : Syn.term -> Sem.answer *)
fun evaluate t
  = iterate (Decomposition.decompose (Sem.CLO_GND (t, Sem.env_init)),
             Sem.sto_init,
             Perm.init)
```

## 6 Refocusing for Reduction-Free Evaluation

We use Danvy and Nielsen's refocusing technique to mechanically transform the iteration of one-step reduction implemented in Section 5.4 into an abstract machine. In this section we show the main steps of this transformation and their effect on the Core Scheme calculus of closures.

The reduction sequence as described in Section 5 consists in repeating the following steps: decomposing a closure into a potential redex and a context, contracting the redex when it is an actual one, and recomposing the context with the contractum, thereby obtaining the next closure in the reduction sequence. The recomposition operation creates an intermediate closure (the next one in the reduction sequence) which is then immediately decomposed in the next iteration. Using refocusing, we can bypass the creation of intermediate closures and proceed directly from one redex to the next. The method is based on the observation that the composition of functions `recompose` and `decompose` can be replaced by a more efficient function, called `refocus`, which is extensionally equal to (and optimally implemented by) `decompose_clo`. The situation is depicted in the following diagram:



### 6.1 An Eval/Continue Abstract Machine over Closures

First, we fuse the functions `recompose` and `decompose` into one function `refocus` that given a closure and its surrounding context, searches for the next redex according to the given reduction strategy. The result is a small-step state-transition system, where `refocus` performs a single transition to the next redex site, if there is one, and `iterate` implements its iteration (after performing the contraction).

Next, we distribute the calls to `iterate` in the definition of `refocus` in order to obtain a big-step state-transition system [11]. The difference between the big-step and the small-step transition system is that in the former, the function `refocus` does not stop on encountering a redex site; it calls the function `iterate` directly. The resulting big-step transition system is presented in Figs. 6 and 7, where `refocus_clo` is an alias for the `refocus` function described above. (The definition of `refocus_clo` and `refocus_cont` is a clone of the definition of `decompose_clo` and `decompose_cont` in Figure 5.)

This resulting transition system is *staged* in that the call to the contraction function is localized in `iterate` whereas `refocus_clo` and `refocus_cont` implement the congruence rules, i.e., the navigation in a closure towards the next redex. Inlining the definition of `iterate` (and thus making do without the data type `decomposition`) yields an eval/continue abstract machine with two mutually recursive transition functions: `refocus_clo` that dispatches on closures, and `refocus_cont` that dispatches on contexts.

---

```

structure EAC_AM = struct
  datatype decomposition = VAL of Sem.value
    | DEC of Redexes.potred * Sem.cont

  (* refocus_clo : Sem.clo * Sem.cont * Sem.sto * Sem.perms -> Sem.answer *)
  fun refocus_clo (Sem.CLO_GND (Syn.QUOTE q, r), rc, s, pg) =
    refocus_cont (rc, Sem.CLO_QUOTE q, s, pg)
  | refocus_clo (Sem.CLO_GND (Syn.VAR i, r), rc, s, pg) =
    iterate (DEC (Redexes.LOOKUP (i, r), rc), s, pg)
  | refocus_clo (Sem.CLO_GND (Syn.LAM (is, t), r), rc, s, pg) =
    iterate (DEC (Redexes.PROC (is, t, r), rc), s, pg)
  | refocus_clo (Sem.CLO_GND (Syn.APP (t, ts), r), rc, s, pg) =
    iterate (DEC (Redexes.PROP_APP (t, ts, r), rc), s, pg)
  | refocus_clo (Sem.CLO_GND (Syn.COND (t0, t1, t2), r), rc, s, pg) =
    iterate (DEC (Redexes.PROP_COND (t0, t1, t2, r), rc), s, pg)
  | refocus_clo (Sem.CLO_GND (Syn.SET (i, t), r), rc, s, pg) =
    iterate (DEC (Redexes.PROP_SET (i, t, r), rc), s, pg)
  | refocus_clo (v as Sem.CLO_QUOTE _, rc, s, pg) =
    refocus_cont (rc, v, s, pg)
  | refocus_clo (v as Sem.CLO_LAM _, rc, s, pg) =
    refocus_cont (rc, v, s, pg)
  | refocus_clo (Sem.CLO_APP (c, cs, vs, p), rc, s, pg) =
    refocus_clo (c, Sem.PUSH (cs, vs, p, rc), s, pg)
  | refocus_clo (Sem.CLO_CALL (v, vs), rc, s, pg) =
    refocus_cont (Sem.CALL (vs, rc), v, s, pg)
  | refocus_clo (Sem.CLO_COND (c0, c1, c2), rc, s, pg) =
    refocus_clo (c0, Sem.SELECT (c1, c2, rc), s, pg)
  | refocus_clo (Sem.CLO_SET (i, r, c1), rc, s, pg) =
    refocus_clo (c1, Sem.ASSIGN (i, r, rc), s, pg)
  | refocus_clo (v as Sem.CLO_UNSPECIFIED, rc, s, pg) =
    refocus_cont (rc, v, s, pg)
  | refocus_clo (v as Sem.CLO_UNDEFINED, rc, s, pg) =
    refocus_cont (rc, v, s, pg)
  | refocus_clo (v as Sem.CLO_PRIMOP (Sem.CWCC, _), rc, s, pg) =
    refocus_cont (rc, v, s, pg)
  | refocus_clo (v as Sem.CLO_CONT _, rc, s, pg) =
    refocus_cont (rc, v, s, pg)

  (* refocus_cont : Sem.cont * Sem.value * Sem.sto * Sem.perms -> Sem.answer *)
  and refocus_cont (Sem.HALT, v, s, pg) =
    iterate (VAL v, s, pg)
  | refocus_cont (Sem.SELECT (c1, c2, rc), c, s, pg) =
    iterate (DEC (Redexes.COND (c, c1, c2), rc), s, pg)
  | refocus_cont (Sem.ASSIGN (i, r, rc), c, s, pg) =
    iterate (DEC (Redexes.UPDATE (i, r, c), rc), s, pg)
  | refocus_cont (Sem.PUSH (nil, vs, p, rc), v, s, pg) =
    iterate (DEC (Redexes.UNPERMUTE (v, vs, p), rc), s, pg)
  | refocus_cont (Sem.PUSH (c :: cs, vs, p, rc), v, s, pg) =
    refocus_clo (c, Sem.PUSH (cs, v :: vs, p, rc), s, pg)
  | refocus_cont (Sem.CALL (vs, rc), v, s, pg) =
    iterate (DEC (Redexes.BETA (v, vs), rc), s, pg)

  (* iterate : decomposition * Sem.sto * Sem.perms -> Sem.answer *)
  and iterate ... =
    ...

  (* evaluate : Syn.term -> Sem.answer *)
  fun evaluate t
    = refocus_clo (Sem.CLO_GND (t, Sem.env_init),
      Sem.HALT,
      Sem.sto_init,
      Perm.init)
end

```

---

**Fig. 6.** Staged eval/apply/continue abstract machine over closures (part 1/2)

```

and iterate (VAL v, s, pg) =
  Sem.VALUE (v, s, pg)
| iterate (DEC (Redexes.LOOKUP (i, r), rc), s, pg) =
  (case Env.lookup (i, r)
   of (SOME l)
     => (case Sto.fetch (l, s)
          of (SOME v)
            => (case v
                  of Sem.CLO_UNDEFINED
                    => Sem.STUCK "undefined value"
                  | _
                    => refocus_cont (rc, v, s, pg))
      | NONE
        => Sem.STUCK "attempt to read an invalid location")
   | NONE
     => Sem.STUCK "attempt to reference an undeclared variable")
| iterate (DEC (Redexes.UNPERMUTE (v, vs, pi), rc), s, pg) =
  refocus_clo (Sem.CLO_CALL (pi (v, vs)), rc, s, pg)
| iterate (DEC (Redexes.BETA (Sem.CLO_LAM (is, t, r, 1), vs), rc), s, pg) =
  if List.length is = List.length vs
  then let val (ls, s') = Sto.news (s, vs)
    in refocus_clo (Sem.CLO_GND (t, Env.extends (is, ls, r)), rc, s', pg)
  end
  else Sem.STUCK "arity mismatch"
| iterate (DEC (Redexes.BETA (Sem.CLO_PRIMOP (Sem.CWCC, _), vs), rc), s, pg) =
  if 1 = List.length vs
  then let val (l, s') = Sto.new (s, Sem.CLO_UNSPECIFIED)
    in refocus_clo (Sem.CLO_CALL (hd vs, [Sem.CLO_CONT (rc, l)]), rc, s', pg)
  end
  else Sem.STUCK "arity mismatch"
| iterate (DEC (Redexes.BETA (Sem.CLO_CONT (rc', 1), vs), rc), s, pg) =
  if 1 = List.length vs
  then refocus_cont (rc', hd vs, s, pg)
  else Sem.STUCK "arity mismatch"
| iterate (DEC (Redexes.BETA (_ , vs), rc), s, pg) =
  Sem.STUCK "attempt to apply a non-procedure"
| iterate (DEC (Redexes.UPDATE (i, r, v), rc), s, pg) =
  (case Env.lookup (i, r)
   of (SOME l)
     => (case Sto.update (l, v, s)
          of (SOME s')
            => refocus_cont (rc, Sem.CLO_UNSPECIFIED, s', pg)
      | NONE
        => Sem.STUCK "attempt to write an invalid location")
   | NONE
     => Sem.STUCK "attempt to assign an undeclared variable")
| iterate (DEC (Redexes.COND (Sem.CLO_QUOTE (Syn.QBOOL false), c1, c2), rc),
          s, pg) =
  refocus_clo (c2, rc, s, pg)
| iterate (DEC (Redexes.COND (_ , c1, c2), rc), s, pg) =
  refocus_clo (c1, rc, s, pg)
| iterate (DEC (Redexes.PROC (is, t, r), rc), s, pg) =
  let val (l, s') = Sto.new (s, Sem.CLO_UNSPECIFIED)
  in refocus_cont (rc, Sem.CLO_LAM (is, t, r, 1), s', pg) end
| iterate (DEC (Redexes.PROP_APP (t, ts, r), rc), s, pg) =
  let val ((pi, rev_pi_inv), pg') = Perm.new pg
    val (c, cs) = rev_pi_inv (Sem.CLO_GND (t, r),
                                map (fn t => Sem.CLO_GND (t, r)) ts)
    in refocus_clo (Sem.CLO_APP (c, cs, nil, pi), rc, s, pg') end
| iterate (DEC (Redexes.PROP_COND (t0, t1, t2, r), rc), s, pg) =
  refocus_clo (Sem.CLO_COND (Sem.CLO_GND (t0, r),
                            Sem.CLO_GND (t1, r),
                            Sem.CLO_GND (t2, r))), rc, s, pg)
| iterate (DEC (Redexes.PROP_SET (i, t, r), rc), s, pg) =
  refocus_clo (Sem.CLO_SET (i, r, Sem.CLO_GND (t, r)), rc, s, pg)

```

**Fig. 7.** Staged eval/apply/continue abstract machine over closures (part 2/2)

## 6.2 An Abstract Machine over Terms and Environments

The result of applying refocusing to the calculus of closures is a machine operating on closures, as shown in Section 6.1. Since we are not interested in modeling the execution of programs in the closure calculus, but in Core Scheme, i.e., with explicit terms and environments, we go the rest of the way and bypass closure manipulation using the method developed in our previous work [3, 4, 9].

To this end, we first short-circuit the ‘corridor’ transitions corresponding to building intermediate closures – these are the transitions corresponding to the propagation of environments in closures: specifically, we observe that each of the closures built with CLO\_COND, CLO\_APP and CLO\_SET is immediately consumed in exactly one of the clauses of refocus after being constructed. Since these closures were only needed to express intermediate results of one-step reduction

```

structure EC_AM = struct
  type env = Sto.loc Env.env

  datatype primop = CWCC

  datatype clo = CLO_GND of Syn.term * env

  datatype value = VAL_QUOTE of Syn.quotation
    | VAL_UNSPECIFIED
    | VAL_LAM of (ide list * Syn.term) * env * Sto.loc
    | VAL_PRIMOP of primop * Sto.loc
    | VAL_CONT of cont * Sto.loc
  and     cont = HALT
    | SELECT of clo * clo * cont
    | ASSIGN of ide * env * cont
    | PUSH of clo list * value list * value Perm.perm * cont
    | CALL of value list * cont

  local val (l_primop, s1) = Sto.new (Sto.empty, VAL_UNSPECIFIED)
        val (l_cwcc, s2) = Sto.new (s1, VAL_PRIMOP (CWCC, l_primop))
  in val env_init = Env.extend ("call/cc", l_cwcc, Env.empty)
      val sto_init = s2
  end

  type sto = value Sto.sto

  type perms = (value, clo) Perm.permgen

  datatype answer = VALUE of value * sto * perms
    | STUCK of string

(* eval : Syn.term * env * cont * sto * perms -> answer *)
fun eval ... =
  ...
(* continue : cont * value * sto * perms -> answer *)
and continue ... =
  ...
(* evaluate : Syn.term -> answer *)
fun evaluate t =
  eval (t, env_init, HALT, sto_init, Perm.init)
end

```

**Fig. 8.** The eval/continue abstract machine over terms and environments (part 1/3)

---

```

fun eval (Syn.QUOTE q, r, rc, s, pg) =
  continue (rc, VAL_QUOTE q, s, pg)
| eval (Syn.VAR i, r, rc, s, pg) =
  (case Env.lookup (i, r)
   of (SOME l)
     => (case Sto.fetch (l, s)
          of (SOME v)
            => (case v
                  of VAL_UNDEFINED
                    => STUCK "undefined value"
                  | _
                    => continue (rc, v, s, pg))
   | NONE
     => STUCK "attempt to read an invalid location")
   | NONE
     => STUCK "attempt to reference an undeclared variable")
| eval (Syn.LAM (is, t), r, rc, s, pg) =
  let val (l, s') = Sto.new (s, VAL_UNSPECIFIED)
  in continue (rc, VAL_LAM ((is, t), r, l), s', pg) end
| eval (Syn.APP (t, ts), r, rc, s, pg) =
  let val ((pi, rev_pi_inv), pg') = Perm.new pg
    val (CLO_GND (t', r'), cs) = rev_pi_inv (CLO_GND (t, r),
                                              map (fn t => CLO_GND (t, r)) ts)
  in eval (t', r', PUSH (cs, nil, pi, rc), s, pg) end
| eval (Syn.COND (t0, t1, t2), r, rc, s, pg) =
  eval (t0, r, SELECT (CLO_GND (t1, r), CLO_GND (t2, r), rc), s, pg)
| eval (Syn.SET (i, t), r, rc, s, pg) =
  eval (t, r, ASSIGN (i, r, rc), s, pg)

```

---

**Fig. 9.** The eval/continue abstract machine over terms and environments (part 2/3)

(and they do not arise from the Core Scheme term language), we can merge the two clauses of `refocus` for each such closure. We then obtain a machine that operates only on `CLO_GND` closures, which are pairs of terms and environments. Hence, we can unfold a closure `CLO_GND (t, s)` into a term and an environment. (The reader is directed to our previous work for numerous other examples of this derivation [3, 4, 9].) This final machine is displayed in Figs. 8, 9, and 10. It is an eval/continue abstract machine for Core Scheme terms, in which an eval configuration consists of a term, an environment, a context, a store, and a permutation generator, and a continue configuration consists of a context, a value, a store, and a permutation generator.<sup>3</sup> The eval transition function dispatches on the term and the continue transition function on the context.

## 7 Analysis

Compared to Figs. 8, 9, and 10, Clinger's machine [5, Fig. 5] has one configuration and two transition functions. This single configuration is a tuple and it is, so to speak, the superposition of our two configurations.

---

<sup>3</sup> This machine is isomorphic to the one in the companion paper [10]. As pointed out there, it is in defunctionalized form: refunctionalizing it yields the continuation-passing evaluation function of a natural semantics, and closure-unconverting this evaluation function yields the compositional valuation function of a denotational semantics.

---

```

and continue (HALT, v, s, pg) =
  VALUE (v, s, pg)
| continue (SELECT (CLO_GND (t1, r1), CLO_GND (t2, r2), rc),
            VAL_QUOTE (Syn.QBOOL false), s, pg) =
  eval (t2, r2, rc, s, pg)
| continue (SELECT (CLO_GND (t1, r1), CLO_GND (t2, r2), rc),
            _, s, pg) =
  eval (t1, r1, rc, s, pg)
| continue (ASSIGN (i, r, rc), v, s, pg) =
  (case Env.lookup (i, r)
    of (SOME l)
      => (case Sto.update (l, v, s)
        of (SOME s')
          => continue (rc, VAL_UNSPECIFIED, s', pg)
        | NONE
          => STUCK "attempt to write an invalid location")
        | NONE
          => STUCK "attempt to assign an undeclared variable")
  | continue (PUSH (nil, vs, pi, rc), v, s, pg) =
    let val (v', vs') = pi (v, vs)
    in continue (CALL (vs', rc), v', s, pg) end
| continue (PUSH ((CLO_GND (t, r)) :: cs, vs, p, rc), v, s, pg) =
  eval (t, r, PUSH (cs, v :: vs, p, rc), s, pg)
| continue (CALL (vs, rc), VAL_LAM ((is, t), r, l), s, pg) =
  if List.length is = List.length vs
  then let val (ls, s') = Sto.news (s, vs)
    in eval (t, Env.extends (is, ls, r), rc, s', pg) end
  else STUCK "arity mismatch"
| continue (CALL (vs, rc), VAL_PRIMOP (VAL_CWCC, _), s, pg) =
  if 1 = List.length vs
  then let val (l, s') = Sto.new (s, VAL_UNSPECIFIED)
    in continue (CALL ([VAL_CONT (rc, l)]), rc), hd vs, s', pg) end
  else STUCK "arity mismatch"
| continue (CALL (vs, rc), VAL_CONT (rc', l), s, pg) =
  if 1 = List.length vs
  then continue (rc', hd vs, s, pg)
  else STUCK "arity mismatch"
| continue (CALL (vs, rc), _, s, pg) =
  STUCK "attempt to apply a non-procedure"

```

---

**Fig. 10.** The eval/continue abstract machine over terms and environments (part 3/3)

The single real difference between Clinger’s machine and the one of Figs. 8, 9, and 10 is that Clinger’s machine dissociates subterms and the current environment. In contrast, the propagation rules of our calculus of closures ensure that terms and environments stick together at all times.

Ergo, the variant of the  $\lambda\widehat{\rho}$ -calculus presented here aptly accounts for Core Scheme. An obvious next step is to scale this calculus to full Scheme and to compare it with the reduction semantics in the R<sup>6</sup>RS. One could also refocus the reduction semantics of the R<sup>6</sup>RS to obtain the corresponding abstract machine. This abstract machine would then provide a sound alternative semantics for the R<sup>6</sup>RS.

## 8 Conclusion and Perspectives

We have presented a version of the  $\lambda\widehat{\rho}$ -calculus and its reduction semantics, and we have transformed a functional implementation of this reduction semantics into

the functional implementation of an abstract machine. This abstract machine is essentially the same as the abstract machine for Core Scheme presented by Clinger at PLDI'98. The transformations are the ones we have already used in the past to derive other abstract machines from other reduction semantics, or to posit a reduction semantics and verify whether transforming it yields a given abstract machine.

This work is part of a larger effort to inter-derive semantic specifications soundly and consistently.

**Acknowledgments.** The authors are grateful to the anonymous reviewers of the 2008 Workshop on Scheme and Functional Programming and of the Mosses Festschrift for their comments, to Will Clinger for extra precisions, and to Ian Zerny for timely feedback. This work is partly supported by the Danish Natural Science Research Council, Grant no. 21-03-0545. We are happy to dedicate it to Peter Mosses for his 60th birthday, and we wish him many happy returns.

## References

1. Abadi, M., Cardelli, L., Curien, P.-L., Lévy, J.-J.: Explicit substitutions. *Journal of Functional Programming* 1(4), 375–416 (1991); A preliminary version was presented at the Seventeenth Annual ACM Symposium on Principles of Programming Languages (POPL 1990)
2. Biernacka, M.: A Derivational Approach to the Operational Semantics of Functional Languages. PhD thesis, BRICS PhD School, Department of Computer Science, Aarhus University, Aarhus, Denmark (January 2006)
3. Biernacka, M., Danvy, O.: A concrete framework for environment machines. *ACM Transactions on Computational Logic* 9(1), 1–30 (2007); Article #6. Extended version available as the research report BRICS RS-06-3
4. Biernacka, M., Danvy, O.: A syntactic correspondence between context-sensitive calculi and abstract machines. *Theoretical Computer Science* 375(1-3), 76–108 (2007); Extended version available as the research report BRICS RS-06-18
5. Clinger, W.D.: Proper tail recursion and space efficiency. In: Cooper, K.D. (ed.) *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Languages Design and Implementation*, Montréal, Canada, June 1998, pp. 174–185. ACM Press, New York (1998)
6. Curien, P.-L.: An abstract framework for environment machines. *Theoretical Computer Science* 82, 389–402 (1991)
7. Curien, P.-L.: Categorical Combinators, Sequential Algorithms and Functional Programming. In: *Progress in Theoretical Computer Science*. Birkhäuser, Basel (1993)
8. Danvy, O.: An Analytical Approach to Program as Data Objects. DSc thesis, Department of Computer Science, Aarhus University, Aarhus, Denmark (October 2006)
9. Danvy, O.: From reduction-based to reduction-free normalization. In: Koopman, P., Plasmeijer, R., Swierstra, D. (eds.) *AFP 2008. LNCS*, vol. 5832. Springer, Heidelberg (2009)

10. Danvy, O.: Towards Compatible and Interderivable Semantic Specifications for the Scheme Programming Language, Part I: Denotational Semantics, Natural Semantics, and Abstract Machines. In: Palsberg, J. (ed.) *Mosses Festschrift*. LNCS, vol. 5700, pp. 162–185. Springer, Heidelberg (2009)
11. Danvy, O., Millikin, K.: On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion. *Information Processing Letters* 106(3), 100–109 (2008)
12. Danvy, O., Millikin, K.: A rational deconstruction of Landin’s SECD machine with the J operator. *Logical Methods in Computer Science* 4(4:12), 1–67 (2008)
13. Danvy, O., Nielsen, L.R.: Refocusing in reduction semantics. Research Report BRICS RS-04-26, Department of Computer Science, Aarhus University, Aarhus, Denmark (November 2004); A preliminary version appeared in the informal proceedings of the Second International Workshop on Rule-Based Programming (RULE 2001), *Electronic Notes in Theoretical Computer Science*, vol. 59.4
14. Felleisen, M.: The Calculi of  $\lambda$ -v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages. PhD thesis, Computer Science Department, Indiana University, Bloomington, Indiana (August 1987)
15. Landin, P.J.: The mechanical evaluation of expressions. *The Computer Journal* 6(4), 308–320 (1964)
16. Marlow, S., Peyton Jones, S.L.: Making a fast curry: push/enter vs. eval/apply for higher-order languages. *Journal of Functional Programming* 16(4–5), 415–449 (2006); A preliminary version was presented at the 2004 ACM SIGPLAN International Conference on Functional Programming (ICFP 2004)
17. Matthews, J., Findler, R.B., Flatt, M., Felleisen, M.: A visual environment for developing context-sensitive term rewriting systems. In: van Oostrom, V. (ed.) *RTA 2004*. LNCS, vol. 3091, pp. 301–311. Springer, Heidelberg (2004)
18. Mosses, P.D.: A foreword to ‘Fundamental concepts in programming languages’. *Higher-Order and Symbolic Computation* 13(1/2), 7–9 (2000)
19. Plotkin, G.D.: Call-by-name, call-by-value and the  $\lambda$ -calculus. *Theoretical Computer Science* 1, 125–159 (1975)
20. Strachey, C.: Fundamental concepts in programming languages. In: International Summer School in Computer Programming, Copenhagen, Denmark (August 1967); Reprinted in *Higher-Order and Symbolic Computation* 13(1/2), 11–49 (2000) with a foreword [18]
21. Strachey, C., Wadsworth, C.P.: Continuations: A mathematical semantics for handling full jumps. Technical Monograph PRG-11, Oxford University Computing Laboratory, Programming Research Group, Oxford, England (1974); Reprinted in *Higher-Order and Symbolic Computation* 13(1/2), 135–152 (2000) with a foreword [22]
22. Wadsworth, C.P.: Continuations revisited. *Higher-Order and Symbolic Computation* 13(1/2), 131–133 (2000)

# Type Checking Evolving Languages with MSOS

M.G.J. van den Brand, A.P. van der Meer, and A. Serebrenik

Technische Universiteit Eindhoven,  
Den Dolech 2, P.O. Box 513, 5600 MB Eindhoven, The Netherlands  
`{m.g.j.v.d.brand,a.p.v.d.meer,a.serebrenik}@tue.nl`

**Abstract.** Evolution of programming languages requires co-evolution of static analysis tools designed for these languages. Traditional approaches to static analysis, e.g., those based on Structural Operational Semantics (SOS), assume, however, that the syntax and the semantics of the programming language under consideration are fixed. Language modification is, therefore, likely to cause redevelopment of the analysis techniques and tools. Moreover, the redevelopment cost can discourage the language engineers from improving the language design.

To address the co-evolution problem we suggest to base static analyses on modular structural operational semantics (MSOS). By using an intrinsically modular formalism, type rules can be added, removed or modified easily. We illustrate our approach by developing an MSOS-based type analysis technique for Chi, a domain specific language for hybrid systems engineering.

## 1 Introduction

Development of a programming language is an ongoing process: new language constructs are being introduced, superficial ones are being removed and semantics of the existing ones is being reconsidered. Traditionally, static analyses are, however, being developed under assumption that the syntax and the semantics of the programming language under consideration are fixed. Traditional semantic specification methods such as Structural Operational Semantics (SOS) encourage this, due to the extensive changes caused by simple additions. Language modification is, therefore, likely to cause redevelopment of the analysis techniques and tools.

The semantic analyses based on the changing semantics should, hence, *co-evolve* together with the language syntax and semantics [1]. Moreover, static semantic analyses should *support decision making* by the language designers by providing them with insights on implications of the decisions taken on the analyses precision and complexity. To this end different design alternatives considered by the language designers should rapidly propagate to semantic analyses. To support both the language/analyses co-evolution and the decision making, we need a flexible formalism for specifying semantic analyses. We believe that modular structured operational semantics (MSOS) [2,3] can be successfully applied to this end. In this paper we validate this claim by implementing a traditional

form of static analysis (type analysis) for a challenging domain-specific language under development (Chi, evolving from Chi 1.0 to Chi 2.0), and verifying that

- (i) the analysis developed is indeed flexible enough to co-evolve with the changing language, and
- (ii) the prototype type checker required by the language designers can be implemented with a limited development effort, i.e., our type checker can support the decision making by the designers.

The programming language we consider in this paper is Chi [4,5], a specification language for hybrid systems. Hybrid systems combine components exhibiting discrete behaviour and components exhibiting continuous behaviour, e.g., a discrete controller and a controlled physical device, such as a fire alarm or a storm surge barrier. Chi is based on the solid mathematical theory of process algebras and hybrid automata [6]. This theory has been successfully applied in industrial projects [7]. From the language perspective, Chi combines features of traditional procedural programming languages (e.g., loops, assignments) with domain-specific elements pertaining to continuity and parallelism.

*Example 1.* To illustrate Chi 2.0<sup>1</sup> consider the following fragment:

```

1  proc ContrivedBuffer<T : type>(chan a?, b! : T) =
2    ||  cont  s: real
3      var   x : T
4      ,     xs : [T] = []
5    ::   eqn   s = SIN(time) + 3
6    ||
7      *(    LEN(xs) < s → a?x ;
8             xs := xs ++ [x]
9            |    LEN(xs) > 0 → b!HD(xs) ;
10           xs := TL(xs)
11         )
12  ]

```

The fragment above illustrates a Chi 2.0 process definition. Structural language elements, e.g., repetition, parallel, alternative or sequential composition are typeset in the bold face, e.g., **\***( ... ), **||**, **|** and **;**, respectively. The built-in functions are typeset in small capitals: LEN, HD, TL and SIN.

Processes are used in Chi 2.0 to group behaviour, in such a way that it can be instantiated and reused multiple times. Process ContrivedBuffer specifies the behaviour of a buffer that receives values of some type  $T$  through channel  $a$  and sends them on through channel  $b$ . Angular brackets in the process definition indicate its polymorphic nature: ContrivedBuffer can be instantiated to manipulate data of any type  $T$ .

In lines 2–4 local variables are declared. The maximum size of the buffer is controlled by the continuous variable  $s$ , varying over time. The list variable  $xs$

---

<sup>1</sup> For the sake of simplicity we opt for the Chi 2.0 notation in the examples.

stores the buffer values, and  $x$  stores temporarily a value just received or to be sent. Lines 5–11 describe the actual behaviour of the process. The behaviour consists of two parallel parts, combined by  $\parallel$ . The equation in line 5 determines the value of the continuous variable  $s$  as a function of a global *time*. The read-only variable *time* is part of the semantics of Chi 2.0. It is updated by the simulation environment and provides access to the current simulation time. The equation should be satisfied at all times during the process execution.

The lines 7–11 specify the discrete behaviour of the process. The behaviour consists of the repeated choice between two conditional alternative steps: receive (denoted  $?$ ) the value  $x$  followed by appending  $x$  to  $xs$ , and send (denoted  $!$ )  $HD(x)$  followed by removing the value  $HD(x)$  from  $xs$ . If both alternatives can be carried out, the process non-deterministically chooses between them.  $\square$

Chi specifications are usually being developed by mechanical engineers, often with limited training in formal methods or software development. As such, the Chi language designers aim at providing the specification developers with as much feedback on potential errors as possible as early in the development life cycle. Since at the early development stages specifications can be incomplete, non-compilable or non-executable, we focus on static semantic analysis, i.e., reasoning about presence or absence of errors without execution or simulation of the actual program.

Specifically, in this paper we focus on type analysis. Type analysis aims at associating each program expression with a set of possible values, known as a *type*. Chi employs prescriptive typing, i.e., types are provided by the developer while the type checker verifies whether the types are consistently used. For instance, expression  $2 + \text{"Hello!"}$  should be rejected by a Chi type checker since addition cannot be applied to a number and a string. Expression  $2 + x$  should be associated, however, with **int** as long as  $x$  itself is associated with **int**.

*Example 2.* Consider the following Chi 2.0 expression:  $\text{TAKE}([y > z, \text{true}] + +xs, \text{size}+1)$ . Here, the *TAKE* function takes a list, say  $l$ , and a natural number, say  $n$ , as parameters and returns a list of the first  $n$  elements of  $l$ , or  $l$  if  $\text{LEN}(l)$  is less or equal than  $n$ . For this expression to be valid in Chi 2.0, the type checker should derive a type for it. In this example, we will discuss how this would be done using a top-down approach.

The top operation of this expression is the invocation of the *TAKE* function. The type of the result of *TAKE* obviously depends on the first parameter, so *TAKE* has an implicit template that specifies this relation. In this example, the value for the list parameter is the result of the concatenation of a list of two booleans, one the result of a comparison and one a constant, and the variable  $xs$ . In Chi 2.0, all elements of a list must be of the same type, hence  $xs$  must be a list of booleans.

The value of the second, numerical, parameter is the result of the addition of the variable *size* and the constant 1. However, to “fit” as parameter of *TAKE* the result must be natural. This means that variable the type of *size* must be natural, because any addition involving integers or reals cannot have a natural result type in Chi 2.0.

Based on the types that the typechecker has derived for the parameters, it has to decide if this invocation of TAKE is valid and if so, determine what the result type is. The function needs a list and a natural number, and if the requirements mentioned above are met, the invocation is correct. The TAKE function has, as mentioned before, an implicit template. More specifically, the return type is a list with elements of the same type as the list provided as parameter. In this case, that means that the result type of the expression is list of booleans.  $\square$

The remainder of the paper is organized as follows. After introducing MSOS and Chi in Sections 2.1 and 2.2, respectively, we review the evolution of typing schemes in Chi (Section 3) and discuss the formalization of type checking rules in MSOS in Section 4. To address (i), while discussing each language construct we stress the similarities and the differences of the formalizations corresponding to different typing schemes implemented or considered during different phases of the evolution of Chi. To address (ii) we have implemented the approach in Maude [8]. We discuss the implementation and lessons learned in Section 5. After reviewing the related work in Section 6 we summarize our contributions and sketch future research directions in Section 7.

## 2 Preliminaries

### 2.1 MSOS

In this section we briefly present the MSOS, modular structural operational semantics, introduced by Peter Mosses in [2,3] to address the modularity shortcomings of SOS [9]. The modularity shortcomings of SOS are related mostly due to difficulty of modifying the environment information: adding a new component, e.g., a stack, to the environment requires modification of all SOS rules. MSOS resolves this by making the structure of the environments implicit. Presenting the formal semantics of MSOS we follow [10,11].

**Definition 1.** (cf. [11]) *A specification in modular structural operational semantics (MSOS) is a structure of the following form.  $\mathcal{M} = \langle \Omega, Lc, Tr \rangle$  where  $\Omega$  is the signature,  $Lc$  is a label category declaration and  $Tr$  is the set of transition rules.*

The *signature* defines function symbols and constants used in the language being specified. *Transition rules* are  $\frac{C}{c}$  intuitively read “whenever all the conditions in  $C$  hold so does the conclusion  $c$ ”, where each condition in  $C$  is either a transition, a predicate or a label expression, and the conclusion  $c$  is a transition. Rules with the empty set of premises  $C$  are called *simple rules*: conclusions of simple rules should always hold.

Transitions are triples that can be interpreted as steps or rewritings:  $e_1 \xrightarrow{\alpha} e_2$  configuration  $e_1$  makes a step (can be rewritten) to configuration  $e_2$  with the label  $\alpha$ . *Label transformers* are ways to modify the labels, while labels in MSOS are morphisms associated with a given category. Recall that *category* is a mathematical construction consisting of a class of objects  $O$ ; a class of morphisms

$A$  between the objects, such that each morphism in  $a \in A$  has a unique source object  $\text{pre}(a)$  and target object  $\text{post}(a)$  in  $O$ ; an associative partial composition function  $\circ$  on the pairs of morphisms; and a function  $\mathbb{1}$  mapping objects to identity morphisms, such that for any  $a \in A$ ,  $\mathbb{1}_{\text{pre}(a)} \circ a = a = a \circ \mathbb{1}_{\text{post}(a)}$ . Formal syntax of the transition rules and label transformers can be found in [11].

*Example 3.* Consider the following rules from [11].

- The rule 
$$\frac{n = n_1 + n_2}{n_1 + n_2 \xrightarrow{\iota} n}$$
 says that if  $n$  is equal to the sum of  $n_1$  and  $n_2$ , then the term  $n_1 + n_2$  can be rewritten to  $n$ . In this case, the premise is a predicate and the conclusion is a transition, labelled with a special silent or unobservable label  $\iota$ . The label  $\iota$  satisfies  $\iota \in \mathbb{1}$ , where  $\mathbb{1}$  is the trivial category consisting of just one object and one morphism.
- The rule 
$$\frac{e_1 \xrightarrow{\alpha} e'_1 \quad e_1 + e_2 \xrightarrow{\alpha} e'_1 + e_2}{e_1 + e_2 \xrightarrow{\alpha} e'_1 + e_2}$$
 reads “whenever  $e_1$  can be rewritten to  $e'_1$  with respect to  $\alpha$ , the summation term  $e_1 + e_2$  can be rewritten to  $e'_1 + e_2$  with respect to the same label. Both the condition and the conclusion are transitions.
- Finally, 
$$\frac{\alpha' = \mathbf{set}(\alpha, \text{env}, \mathbf{get}(\alpha, \text{env})[x \rightarrow v]) \quad e \xrightarrow{\alpha'} e'}{\lambda x(e)v \xrightarrow{\alpha} e'}$$
 allows us to introduce  $x$ . The first condition is a label expression based on two “built-in” functions **set** and **get**. Label  $\alpha'$  is, therefore, obtained from  $\alpha$  by updating the value of **env** to include the mapping of the variable  $x$  to the type  $v$ . The second condition and the conclusion are transitions.

In the rules above, the label category  $Lc$  is **ContextInfo**( $\text{env}$ , *Environment*)( $\mathbb{1}$ ), where **ContextInfo**( $\text{env}$ , *Environment*) is a label transformer derived from a set of environments *Environment*. Hence, e.g., **get**( $\alpha$ ,  $\text{env}$ ) denotes the environment corresponding to the label  $\alpha$ .

Semantics of MSOS is given by means of a mapping to an arrow labelled transition system. Details of the mapping can be found in [11]. In particular, predicates are assumed to be available for use, and not part of the MSOS rules.

## 2.2 Hybrid Specification Language Chi 2.0

In this section we briefly present the syntax of Chi 2.0 [5]. For the sake of brevity and without loss of generality we slightly simplified the syntax. A Chi 2.0 *model* is composed of comma separated series of variable, channel and action declarations followed by **:::** followed by a process description. Variables can be discrete ( $x$  and  $xs$  in Example 1), i.e., their values can change only by means of explicit assignments; continuous ( $s$  in Example 1), i.e., their values are determined by a continuous function of time; and algebraic, i.e., their values may change according to a discontinuous function of time. The change of continuous and algebraic variables can be restricted by a special form of process, called *equation*, e.g.,

**eqn**  $s = \text{SIN}(time) + 3$ . Channels serve for unbuffered, synchronous communication between the processes that can send (!) data to channels or receive (?) data from them. Both variables and channels are *typed*. The atomic types are **bool**, **nat**, **int**, **real**, **string** and **enum** (enumerations). Type constructors operate on existing types to create structured types such as sets, lists, arrays, record tuples, dictionaries, functions, and distributions (for stochastic models). The type corresponding to a channel is the type of data that is communicated via the channel.

Processes can in their turn be atomic or composite. Atomic processes include performing an action, sending or receiving data, assigning values to variables, delaying or performing a blocking conditional action: if the condition does not hold the process will block until the condition becomes true. Furthermore, equations are considered as atomic processes. As seen in Example 1 composite processes can be obtained from the atomic ones by repetitive application of parallel, sequential or alternative composition, or by means of a loop. Furthermore, similarly to models processes can be defined separately, i.e., processes also act as subroutines in Chi 2.0. As such the processes can be *parameterized* by a sequence of parameters (see line 1 in Example 1). Processes can be made even more flexible by the use of *templates* (e.g.,  $\langle T : \text{type} \rangle$  in Example 1).

As we are going to focus on type analysis, we are also interested in built-in functions, which describe mathematical computations without side effects, and operators. These include traditional operators on the booleans, numbers and strings, explicit type conversions, and constructors for composite types, like lists and sets. Similarly to processes functions can make use of templates. *Explicit* templates require the user to supply values for the template parameters, which are then used to instantiate the function. Similarly to Example 1 explicit template parameters are listed between angular brackets: **func**  $f\langle T : \text{type} \rangle(\text{val } a : T) \rightarrow T$ . A common use for function templates is to allow a function to be applied to multiple types of parameters:  $f\langle \text{nat} \rangle(7)$ . It should be noted that we write  $f\langle \text{nat} \rangle(7)$  rather than  $f\langle \text{nat} \rangle(7)$  to distinguish between strings representing types in the Chi 2.0 program (“nat”), referred to as type literals, and actual types (**nat**).

In addition to explicit templates, Chi 2.0 supports functions with implicit templates. *Implicit* templates are instantiated during function invocation. In contrast to explicit templates, the values for the implicit template parameters are determined by the type checker, based on the types of the function parameters supplied by the user. Implicit templates are mainly used to allow functions like **HD()** to work for lists with any type of element, while still allowing the type checker to derive a useful return type.

*Example 4.* In the following function definitions we list the implicit type parameters between [ and ].

The function **LEN()** calculates the length of a list: **func**  $\text{LEN}[T : \text{type}](\text{val } xs : [T]) \rightarrow \text{nat}$ . The type of the elements of the list does not influence the return type, which is reflected in the function declaration by the fact that  $T$  is only used once.

Next consider  $\text{HD}()$  defined as  $\text{func } \text{HD}[T : \text{type}](\text{val } xs : [T]) \rightarrow T$ . Observe that  $T$  occurs both in the parameter type and as the return type. The function  $\text{HD}()$  returns the first element of a non-empty list, so here the actual type of the elements of the list does affect the type checking process.

Finally, the definition of  $\text{SORT}$  uses  $T$  to describe types of multiple parameters  $\text{func } \text{SORT}[T : \text{type}](\text{val } xs : [T], f : (T, T) \rightarrow \text{bool}) \rightarrow [T]$ . The first parameter of function  $\text{SORT}$  is a list of type  $T$ , and the second one is a predicate  $f : (T, T) \rightarrow \text{bool}$  imposing a sorting order on the elements of type  $T$ .  $\square$

Functions are first-class citizens in Chi 2.0, a function can, e.g., be passed as a parameter to another expression (function). One construct using functions as parameters is a *fold*. Folds have four parameters:  $\langle f, i \leftarrow I, p(i), c(i) \rangle$ .  $I$  is the iterator generating values and storing one value at a time in  $i$ . Once a value is generated, predicate  $p$  is applied to it, and if  $p(i)$  is *true*, then the auxiliary value  $c(i)$  is computed. Finally, the aggregation function  $f$  is applied to all the auxiliary values computed. Chi 2.0 allows only a restricted number of aggregation functions such as  $+$ ,  $*$ ,  $\text{MAX}$ , to be used in folds.

*Example 5.* Let  $\langle +, i \leftarrow [1, 2, 3, 4], i > 2, i * i \rangle$  be a fold. Then, each value from the list  $[1, 2, 3, 4]$  is in its turn assigned to  $i$ , compared with 2, and for those values larger than 2 the square is computed. Finally, all squares are added. Hence, the value of  $\langle +, i \leftarrow [1, 2, 3, 4], i > 2, i * i \rangle$  is  $0 + 3 * 3 + 4 * 4$ , i.e., 25.  $\square$

### 3 Evolution of Numerical Types in Chi

In order to illustrate how the type checker expressed in MSOS can co-evolve with the language itself we focus on a part of Chi static semantics that significantly evolved between versions: the type system for numerical values. Chi has three types of numerical values: **nat** for natural numbers, **int** for integer numbers and **real** for real numbers. Mathematically, there is an obvious relation  $\mathbb{N} \subset \mathbb{Z} \subset \mathbb{R}$ . This, however, may or may not be reflected in the type system.

Earlier versions of Chi, such as Chi 1.0 [4]<sup>2</sup>, insisted on a strict separation of the numerical types, i.e., not performing the type widening at all. This approach was motivated by the fact that for the specification engineers the natural numbers usually represent quantities, while the real numbers correspond to measurable aspects of physical artifacts such as speed, pressure, or frequency. Moreover, Chi is also used for training system engineering students with no previous programming experience, and that this audience should be made aware of differences between mathematical sets such as  $\mathbb{N}$ ,  $\mathbb{Z}$  and  $\mathbb{R}$  and programming language types **nat**, **int** and **real**.

Strict separation of types in Chi 1.0 implied, unfortunately, that  $\text{SIN}(0)$  cannot be computed as the sine is defined as a **real**  $\rightarrow$  **real** function. Therefore, later on the language designers considered applying full widening on numerical

<sup>2</sup> Though the report was finalized in 2008, initial steps in the development of Chi 1.0 were made in 1995 and the language was in active use from 2005.

types, i.e., assuming **nat**  $\prec$  **int**  $\prec$  **real**. While Chi 1.0 would disallow  $1 + 2.5$  as addition is being applied to arguments of two different numerical types, this version, that we call “Chi 1.5”<sup>3</sup>, would convert 1 to 1.0 and calculate the result. In this way, however, distinction between different numerical types becomes blurred. From a computer science perspective, this is not really significant, but from a hybrid systems viewpoint, different numerical types are used for physically different things, which should not be confused. Therefore, while developing Chi 2.0 [12] the language designers have chosen to apply type widening solely to constants explicitly mentioned in the model. In this way the designers felt the novices will still understand the distinction between mathematical sets and types in Chi 2.0, without being forced to write 0.0 instead of 0 just to pacify the type checker. In order to facilitate this, we created two new types, **cnat** and **cint**, for constant numeric values, to allow a separate treatment. This solution has been reported by the language designers as preferred to both Chi 1.0 and Chi 1.5.

## 4 Typechecking Chi in MSOS

As explained in Section 2.1, MSOS rules describe transitions between configurations of transition systems. A computation, then, is a sequence of transitions from an initial to a final configuration. In the case where a type system has been specified using MSOS, the initial configuration is a program expression to be typechecked, and the desired final configuration is the type corresponding to that expression. In this section, we will describe our definition of the Chi 2.0 type system in MSOS.

While specifying the MSOS transition rules we opt for *big-step semantics*, i.e., we describe how the overall results of the executions are obtained as opposed to the *small-step semantics*, describing how individual steps take place. As recognized by Mosses in [2] big-step semantics is preferable for specifying type-checking. As static semantics in this case involves environments not changed by the type-checking rules in a corresponding MSOS, most labels are identity morphisms. We omit those labels for the sake of readability.

In the remainder of this section we present MSOS rules for three typing schemes considered in Section 3: no widening (Chi 1.0), full widening (“Chi 1.5”) and widening restricted to constants (Chi 2.0). We stress how the rules can express changes in the typing schemes.

### 4.1 Basic Type Rules

The basis of every type system lies in the atomic expressions, such as constants and variable references. To derive types for constants we use simple rules as typing of these constants does not depend on any precondition. Labels in small

---

<sup>3</sup> “Chi 1.5” is a name we use for the sake of convenience rather than an officially released version of Chi.

capitals are used for the ease of reference only and should not be considered as a part of the MSOS syntax:

$$\frac{}{\mathbf{true} \longrightarrow \mathbf{bool}} (\mathbf{TRUE}) .$$

Unfortunately, this approach would require providing a separate transition rule for *each* constant. As this is not feasible for numerical types we introduce unary predicates *natural\_number*, *integer\_number* and *real\_number* that evaluate to *true* according to the raildiagrams of [4]. Formally, *natural\_number* evaluates to *true* if its argument is a digit zero, or a non-empty sequence of digits starting with a non-zero digit; *integer\_number* evaluates to *true* if its first character is + or -, and the remainder is either a digit zero, or a non-empty sequence of digits starting with a non-zero digit. Finally, *real\_number* evaluates to *true* if its argument is composed of two parts divided by a point, and each one of the parts is either a digit zero, or a non-empty sequence of digits starting with a non-zero digit.

Using these predicates for Chi 1.0 and Chi 1.5 we write

$$\frac{\mathbf{natural\_number}(n)}{n \longrightarrow \mathbf{nat}} (\mathbf{NAT}) \quad \frac{\mathbf{integer\_number}(z)}{z \longrightarrow \mathbf{int}} (\mathbf{INT}) \quad \frac{\mathbf{real\_number}(r)}{r \longrightarrow \mathbf{real}} (\mathbf{REAL}) .$$

The typing scheme of Chi 2.0, however, requires refining the type system to include special types for natural (**cnat**) and integer (**cint**) constants, and updating the transition rules as follows:

$$\frac{\mathbf{natural\_number}(n)}{n \longrightarrow \mathbf{cnat}} (\mathbf{NAT}) \quad \frac{\mathbf{integer\_number}(z)}{z \longrightarrow \mathbf{cint}} (\mathbf{INT}) .$$

In order to determine the type of a variable one should consult the environment. Similarly to Example 3 we use the built-in function **get** to derive the current environment. Since the environment is a morphism it can be applied to the variable name to obtain the variable's type:

$$\frac{\mathbf{get}(\alpha, \mathbf{env})(x) = v}{x \xrightarrow{\alpha} v} (\mathbf{LOOKUP}) .$$

If the variable name  $x$  is not included in the current environment  $\mathbf{get}(\alpha, \mathbf{env})$ , the type derivation fails. Otherwise, the lookup function gets the appropriate type  $v$ . Obviously, since LOOKUP does not mention numerical types explicitly, this rule is affected by the different typing schemes.

## 4.2 Overloading

Numerical operators can be often applied to *different* sets of types of arguments, e.g., + is used to denote addition of two natural numbers, two integers, etc. This form of polymorphism is known as *overloading*. Overloading can be seen either as different ways to invoke the same operator, or as invocations of different operators with the same name. In the latter case, the choice is made based on the

data types of the parameters passed. The type checker should distinguish between different ways the same operator is invoked or between different operators of the same name, and derive an appropriate type for the result of the operator application.

Considering an overloaded operator as a number of different operators allows to express the typing rules for each one of them separately. For instance, addition of two integers can be expressed as  $\frac{e_1 \rightarrow \text{int} \quad e_2 \rightarrow \text{int}}{e_1 + e_2 \rightarrow \text{int}}$ . This solution

requires, however, a separate rule for each numerical type, e.g., 3 rules for the no widening typing scheme of Chi 1.0 and 9 rules for the full widening scheme of “Chi 1.5”. In general, this solution defeats the type checker flexibility we aim at: addition of a new numerical type requires adding  $2n$  new transition rules, where  $n$  is the current number of numerical types.

To address this problem we need a more compact way of expressing the relation between different types. Specifically, for addition we need predicates “numerical”, restricting the application of  $+$  only to numerical types, and “lub” allowing to choose a the resulting type based on the types of the arguments:

$$\frac{\begin{array}{c} e_1 \rightarrow t_1 \quad e_2 \rightarrow t_2 \\ \text{numerical}(t_1) \quad \text{numerical}(t_2) \quad \text{lub}(t_1, t_2, t_3) \end{array}}{e_1 + e_2 \rightarrow t_3} \text{ (ADD)} .$$

Type widening is explicitly present in the rule for addition in the form of the “lub” predicate, least upper bound with respect to the  $\preceq$  relation on numerical types. In the case of no widening (Chi 1.0), the relation  $\prec$  is empty and  $\text{lub}(t_1, t_2, t_3)$  is true if  $t_1$ ,  $t_2$  and  $t_3$  coincide. For full widening (“Chi 1.5”)  $\preceq$  is a linear ordering and  $t_3$  is the larger one of  $t_1$  and  $t_2$ . Finally, Chi 2.0 implies **cnat**  $\prec$  **cint**, **cnat**  $\prec$  **nat**, **cint**  $\prec$  **int**, **cnat**  $\prec$  **int**, **cnat**  $\prec$  **real** and **cint**  $\prec$  **real**. The relation  $\preceq$  is a partial ordering and  $t_3$  should be the least upper bound of  $t_1$  and  $t_2$ :  $\text{lub}(\text{nat}, \text{cint}, \text{int})$  should evaluate to *true*.

### 4.3 Type Checking User-Defined Functions

In addition to operators similar to those discussed in Section 4.1 and 4.2 Chi offers a number of functions, e.g., **HD()**, **TL()** and **LEN()** from Example 1. Moreover, unlike operators, new function definitions can be introduced by the user. Hence, it is impossible to have a separate rule for each function, like we have for each operator, and a more generic solution is sought. This generic solution should support (1) functions with an arbitrary number of parameters, (2) any combination of parameter types, as well as (3) explicit and (4) implicit templates as mentioned in Section 2.2. We will first discuss the functions without templates, postponing the discussion of explicit and implicit templates until Sections 4.4 and 4.5, respectively.

In addition to function invocation present in traditional programming languages, Chi supports variables of *function type*. Expressions can be assigned to these variables as long as the type of the expression is a function type.

Moreover, these expressions, known as function expressions, can be used similarly to traditional function invocations.

*Example 6.* Consider the following example (adapted from [4]):

```

1  func f(val x : int) → int = [ret x * x]
2  model M() =
3  [[      var      p : (int) → int
4        ,         fv, pv : int
5        ::          p := f
6        ;         fv := f(1)
7        ;         pv := p(-1)
8
9    ]]

```

Line 7 shows the traditional function invocation:  $f$  is applied to 1 to compute the value to be assigned to  $fv$ . In Line 6  $f$  is assigned to  $p$  allowing the developer to use  $p$  similarly to  $f$  (Line 8).  $\square$

Intuitively, to type check  $p(-1)$  correctly in Example 6 we proceed in four steps. First we need to provide an appropriate typing for  $p$ , which in this case would be  $\mathbf{int} \rightarrow \mathbf{int}$ . Then to derive the type for  $-1$ , i.e.,  $\mathbf{int}$  ( $\mathbf{cint}$ ). If the full widening (“Chi 1.5”) or constant widening (Chi 2.0) typing system are used we might need to widen the argument types before applying the function, types of the arguments should *match* the function signature but not necessarily be identical to it. Finally, we report  $\mathbf{int}$  as the type of  $p(-1)$ .

The MSOS rule FUNCTION INVOCATION given below formalizes this intuition. Note that  $*$  denotes a sequence of objects of the same kind: if  $t$  is a type, then  $t^*$  is a sequence of types; if  $e$  is an expression, then  $e^*$  is a sequence of expressions, etc.

$$\frac{e_1 \longrightarrow (\text{func}(t_1^*) \rightarrow t_2) \quad e_p^* \longrightarrow t_p^* \quad \text{widen}(t_p^*, t_1^*)}{e_1(e_p^*) \longrightarrow t_2} \text{ (FUNCTION INVOCATION)}$$

The first precondition of FUNCTION INVOCATION corresponds to the first step:  $e_1$  has to evaluate to some function type in order for invocation to be possible. Next, the types of the parameters are determined and widened to types appearing in the function type of  $e_1$ . If these preconditions hold, we can report that the type derived for the function invocation is the one prescribed by the function type of  $e_1$ .

Formally, widen applied to two sequences of equal length  $t_p^*$  and  $t_1^*$  is evaluated to *true* if each element in  $t_p^*$  is not larger with respect to  $\prec$  than the corresponding element in  $t_1^*$ .

The type widening is explicitly present in the form of the “widen” predicate. In the case of no widening (Chi 1.0), this predicate should describe the identity relation between types, extended point-wise to sequences. In the full widening

and constant widening case, the “widen” predicate has to use the  $\prec$  relation to decide if a certain combination is a correct match. In effect, “widen” serves as a generalization of  $\prec$ , that accepts sequences as arguments instead of single elements.

#### 4.4 Explicit Templates

Recall that in the FUNCTION INVOCATION in Section 4.3, the first step consisted in evaluating the corresponding function expression. In the majority of the cases, a function expression is simply a reference to a declared function or a function variable, but it can also be the instantiation of an explicit template. An explicit function template has a number of template parameters. During template instantiation, the values provided for the parameters are applied to the template, resulting in a new function instance.

$$\frac{e_1 \longrightarrow \text{func}[t_{te}^*, id^*](t_p^*) \rightarrow t_r \\ e_v^* \longrightarrow t_{tv}^* \quad \text{widen}(t_{tv}^*, t_{te}^*) \quad \text{eval}(e_t^*, t_v^*) \\ \text{replace}(id^*, t_v^*, t_p^*, t_{p2}^*) \quad \text{replace}(id^*, t_v^*, t_r, t_{r2})}{e_1 \langle e_v^*, e_t^* \rangle \longrightarrow \text{func}(t_{p2}^*) \rightarrow t_{r2}} \text{ (EXPLICIT TEMPLATE)}$$

Type checking an explicit template instantiation is quite similar to function invocation. The topmost step of the rule consists in deriving the type of  $e_1$  to check that it indeed represents a function template. The next step is to derive and check the types of the template parameters. In this version of the rule, we assume that the type parameters are listed last among the template parameters, and that we can distinguish between the type parameters and non-type parameters. Non-type parameters have no further influence on the type derivation, so no further actions are required.

In contrast, the values of the type parameters influence the type derivation. All expressions used for explicit template parameters are required by the language definition to be compile-time constants, so they can be calculated here. The predicate  $\text{eval}$  maps strings representing types (“nat”) to actual types (**nat**). The predicate  $\text{replace}$  is applied to get new parameter ( $t_{p2}^*$ ) and return ( $t_{r2}$ ) types, where all occurrences of  $id^*$  have been replaced by the corresponding value in  $t_v$ . If that can be done successfully, we can report that the type derived for the function expression is a function with parameter types  $t_{p2}^*$  and return type  $t_{r2}$ .

*Example 7.* Recall the example explicit template function type in 2.2: **func**  $f\langle T : \text{type}\rangle(\text{val } a : T) \rightarrow T$ . In this case, there are no non-type parameters in the template, so those preconditions are trivially true. In the example, the actual instantiation is  $f(\text{nat})$ . The expression **nat** will evaluate to **nat**, which can replace  $T$  in  $(\text{val } a : T) \rightarrow T$  with no problem. The resulting function type is  $(\text{val } a : \text{nat}) \rightarrow \text{nat}$ . The next step is function invocation, described in Section 4.3.

The type widening present here is similar to the function invocation case. Here, only a subset of the parameters has to be widened, because the type parameters

are not numerical and cannot be widened. Apart from that, the widening works exactly the same in each case. Observe that **cnat** and **cint** are not explicitly part of the Chi 2.0 language and do not have a type literal, and, hence, cannot be provided as a type parameter in EXPLICIT TEMPLATE.

#### 4.5 Implicit Templates

In Example 1 we have seen several functions applicable to lists of any type of elements, e.g., **HD()**, **TL()** and **LEN()**. Chi distinguishes between the lists of naturals, lists of booleans, lists of strings, etc. and neither can be considered as a generalization of the other. Hence, if we want the same function to be applicable to all kinds of lists, we would need a separate rule for each one of them. Moreover, providing an element type explicitly would incur an unnecessary overhead on the specification engineer. As this is obviously undesirable, implicit templates were added to the language allowing functions to be defined for variable parameter types without the specification engineer to have name the types explicitly.

The addition of implicit templates makes the rule for function invocation, presented below, significantly more complicated. The function type that results from the function expression can no longer directly be invoked. First we need to determine to which function, represented by metavariable  $e_1$ , is called:  $e_1 \longrightarrow \text{func}[id^*](t_f^*) \rightarrow t_{f2}$ . Next we need to determine types of the actual parameters  $e_p^* \longrightarrow t_p^*$ , so can determine the values of the type parameters with  $\text{unify}(id^*, t_f^*, t_p^*, t_t^*)$ . Finally, we need to replace the type parameters by their values in the return type of the function to which  $e_1$  corresponds.

$$\frac{\begin{array}{c} e_1 \longrightarrow \text{func}[id^*](t_f^*) \rightarrow t_{f2} \\ e_p^* \longrightarrow t_p^* \\ \text{unify}(id^*, t_f^*, t_p^*, t_t^*) \quad \text{replace}(id^*, t_t^*, t_{f2}, t_r) \\ e_1(e_p^*) \longrightarrow t_r \end{array}}{e_1(e_p^*) \longrightarrow t_r} \text{ (IMPLICIT TEMPLATE)}$$

Predicate  $\text{unify}(id^*, t_f^*, t_p^*, t_t^*)$  is true if there exists a widening of  $t_f^*$  unifiable with  $t_p^*$  and  $t_t^*$  corresponds to  $id^*$  after the unification. Predicate  $\text{replace}(id^*, t_t^*, t_{f2}, t_r)$  is true if  $t_r$  can be obtained from  $t_{f2}$  by replacing type variables from  $id^*$  by the respective types from  $t_t^*$ .

This rule is the most affected one by type widening. The majority of the widening takes place in the “unify” predicate. If there is no widening, “unify” can match types and set values for type variables directly. If there is widening, care has to be taken to ensure that the value for the type variables is not chosen too soon, which could incorrectly fail to derive a type for a correct expression. In addition, if the restricted widening is in effect, a step has to be added to prevent the possibility of a function returning a result of type **cnat** or **cint**, even if all parameters are of those types. A possible solution is the introduction of a “widenConst” predicate:

$$\frac{\begin{array}{c} e_1 \longrightarrow \text{func}[id^*](t_f^*) \rightarrow t_{f2} \\ e_p^* \longrightarrow t_p^* \quad \text{unify}(id^*, t_f^*, t_p^*, t_t^*) \\ \text{replace}(id^*, t_t^*, t_{f2}, t_{r2}) \quad \text{widenConst}(t_{r2}, t_{r3}) \\ e_1(e_p^*) \longrightarrow t_{r3} \end{array}}{e_1(e_p^*) \longrightarrow t_{r3}} \text{ (IMPLICIT TEMPLATE)}$$

The “widenConst” predicate holds if  $t_{r2}$  is equal to  $t_{r3}$ , with all instances of **cnat** replaced by **nat** and all instances of **cint** replaced by **int**. This guarantees that a function always returns the proper type.

## 4.6 Folds

Folds in Chi exist to allow simple, often occurring loops to be expressed in one straightforward expression or statement. In this section we discuss addition fold expressions, i.e., folds having  $+$  as the aggregation function.

Type checking a fold expression roughly follows the steps of the fold evaluation discussed in Section 2.2. A fold expression has three subexpressions, referred to as  $e_1$ ,  $e_2$  and  $e_3$ . First, the type  $t_{id}$  of the generator expression  $e_1$  is determined. The result has to be some kind of container, e.g. a list, and the predicate *container* expresses the fact that  $t_{id}$  is the type of the elements in the container type  $t_1$ . The identifier  $id$  is then temporarily added to the environment as a variable with type  $t_{id}$  (cf. Example 3). The types of guard  $e_2$  and transformer  $e_3$  are derived in this new environment. Finally, the results of applying  $e_3$  to each value will have to be added together, so they have to meet the criteria for addition. In this case, we know the types of all values will be the same, so the “maximum” predicate is not needed, only “numerical”. Type widening is not relevant here.

$$\frac{\begin{array}{c} \text{container}(t_1, t_{id}) \quad e_1 \xrightarrow{\alpha} t_1 \\ \alpha' = \mathbf{set}(\alpha, \text{env}, \mathbf{get}(\alpha, \text{env})[id \rightarrow t_{id}]) \\ e_2 \xrightarrow{\alpha'} \mathbf{bool} \quad e_3 \xrightarrow{\alpha'} t_4 \quad \text{numerical}(t_4) \end{array}}{\langle +, id \leftarrow e_1, e_2, e_3 \rangle \xrightarrow{\alpha} t_4} \text{ (ADDITION FOLD)}$$

*Example 8.* In Example 5 we have introduced the following addition fold:  $\langle +, i \leftarrow [1, 2, 3, 4], i > 2, i * i \rangle$ . Here, the  $id$  is  $i$ , and the collection  $e_1$  is  $[1, 2, 3, 4]$ ,  $e_2$  is  $i > 2$  and  $e_3$  is  $i * i$ . First, we determine the type  $t_1$  of  $e_1$  to be a list of **nat**. This means that  $t_{id}$  is determined to be **nat**, and the environment is (temporarily) updated to register that  $i$  is a natural number. Though the specific rules are not detailed in this paper, it should be obvious that  $i > 2$  and  $i * i$  evaluate in the enhanced environment to **bool** and **nat**, respectively. This means the fold is valid, and the result type  $t_4$  can be concluded to be **nat**.  $\square$

To adapt ADDITION FOLD to the Chi 2.0 typing scheme we need to use the same idea as in IMPLICIT TEMPLATE: widenConst should be applied to the resulting type.

## 4.7 Evolution: Summary

As mentioned in the introduction Chi is an evolving language and the typing system of Chi evolves together with language. In this section we have reviewed three different typing systems considered by the language designers: not using type widening at all (Chi 1.0), using full widening (“Chi 1.5”) and using a

restricted from of widening, pertaining solely to explicitly mentioned constants. We have seen that the only parts of the MSOS transition rules affected by change of the typing system are related to auxiliary predicates implementing widening or related notions such as replacement.

## 5 Prototype Implementation

To assess the feasibility of the MSOS-based approach as well as to provide the language developers with insights in different alternatives considered, we have developed a prototype implementation. We base our prototype implementation of the type checker on the MMT [11], an interpreter for MSOS specifications in Maude [8].

In the notation used by MMT transition  $e \xrightarrow{\alpha} t$  is written as  $E =\{\alpha\} \Rightarrow T$ . Specifically, if  $\alpha$  is  $i$  we write  $E =\{\dots\} \Rightarrow T$ . In theory, MMT supports predicates, but in practice, we often found it easier to rework predicates to transitions. For example, the rule for function invocation as discussed in Section 4.3 becomes:

```
(Exp1 =\{\dots\}=> (Exp3 , func(Type1*,Type2))) ,
((Exp*) match (Type1*) as (Type2)) =\{\dots\}=> Type3
-- -----
(Exp1 calls (Exp*)) : Exp =\{\dots\}=>
    (((Exp3 , func(Type1*,Type2)) calls (Exp*)), Type3) .
```

In this implementation, the first line corresponds to  $e_1 \longrightarrow (\text{func}(t_1^*) \rightarrow t_2)$  in the original rule FUNCTION INVOCATION. A notable difference is that in the MMT implementation we choose to keep the expression together with the types, in order to get a clearer view of exactly which steps the system made to reach its conclusion. The second line corresponds to  $e_p^* \longrightarrow t_p^*$  and  $\text{widen}(t_p^*, t_1^*)$ . We combined the type derivation step with the predicate widen into one expression, that rewrites to the return type of the function if the invocation is valid.

Using the machinery provided by MMT and Maude the type checker for a representative subset of Chi 2.0 was implemented by the second author with no previous Maude experience in less than one month time. A *representative subset* of Chi 2.0 was chosen jointly with the language designers. It omits some similar operators, e.g., disjunction and  $\leq$  while conjunction and  $\geq$  have been included.

Developing the prototype type checker implementation allowed us to provide the language developers with a number of suggestions on how the language can be further improved. These improvement suggestions have been accepted by the language developers and will become part of the coming release of Chi. Some of our improvement suggestions pertained to:

- return types for built-in functions, e.g., MAX has been redefined to **(nat, int) → nat** instead of **(nat, int) → int**, stated in [12];
- syntax readability, e.g., consistency of naming conventions for built-in operators;
- type literals as the first-class citizens, i.e., one can also write **var t : type = nat;**

- anonymous functions, i.e.,  $\lambda$ , currently being “hidden” inside the fold construct;
- generalization of the fold constructs including user-defined (anonymous) aggregation functions.

## 6 Related Work

In this paper we focused on co-evolution of static semantic checkers, in our case, a type checker, and prototyping for developing languages, e.g., the academic ones. The idea of using structured operational semantics for type checking is, however, not new and goes back to [9,13]. More recent work on this subject can be found, e.g., in [14] and [15]. These works, however, assume the programming language being analyzed, to be fixed and do not address the co-evolution issues.

Subtypes and type coercion (also known as type widening) have been considered e.g. in [16,17,18]. In [16], an algorithm is discussed for type inference for type systems with subtypes. This algorithm was considered too inefficient for practical use[17][18]. It is also proven that the general problem of type inference with subtypes is PSPACE-complete. This raises the question if practically useable typechecker can be generated from MSOS type specifications, due to the generic nature of MSOS. Experimental evidence of this will be considered an important part of our future work.

To achieve the desired evolvability of the type checker we have chosen MSOS as the basis of our approach. MSOS, modular structural operational semantics, was introduced by Peter Mosses in [3,2]. MSOS has been successfully applied to develop analysis tools for formal properties, e.g., presence of deadlocks or LTL, of a concurrent Caml-like language and Java [19]. The differences between [19] and our work pertain however, not only to the specific language or analysis considered, but also to the theoretical focus on co-evolution. Type-checking with MSOS was also briefly mentioned in [2].

Significant research effort [20,21,22] has been spent on type checking using attribute grammars [23]. Attribute grammars are a well-known semantics specification formalism based on defining attributes for nodes in the syntax tree. Modular extensions of attribute grammars have been considered, e.g., in [24]. MSOS transition rules can be seen as related to attribute grammars with labels considered as inherited, i.e., based on the ancestors, attributes. Unlike attribute grammars, MSOS transition rules do not necessarily follow the structure of the syntax tree. Hence, the MSOS transition rules provide a more declarative perspective, facilitating rapid prototype development essential for the co-evolution.

Another approach to type checking definition is based on algebraic equations. In [25] a type checker for Pascal has been defined using ASF+SDF [26]. ASF+SDF is modular algebraic specification formalism which is supported by an interactive development environment, the ASF+SDF Meta-Environment [27]. The modularity of ASF+SDF provides a nice separation of type checking rules per language construct. However, the rules themselves are rather complicated because of the explicit propagation of environments via the arguments of the

rewrite functions. ASF+SDF has been used to specify the type checker of the previous version of Chi. However, the requirement of evolution of Chi 2.0 in combination with the observed drawback made us decide to explore another approach.

Expressing both the language semantics and the semantic analysis rules in the same semantic framework should lead to seamless integration of the two. The idea of integrating the language semantics and the type checker was pursued, e.g., by Dinesh in [28]. The focus of [28] is, however, on providing the developer with appropriate feedback in case the type checker fails (where did the error occur, what type was expected) rather than flexible type checker specification.

Based on the rules described in Section 4 we have implemented a prototype type checker using MMT. A system similar to MMT was Typol. The Typol semantics specification system [29] is based on inference rules, which can be compiled into Prolog programs. Experiments in Typol discussed in [29] are the implementation of a type checker for the toy language ASPLE [30], a type checker for ML and a type checker for a part of PASCAL. The authors claim that Typol programs can be easily made more incremental, which is a great advantage for a type checker used in an interactive environment. In comparison with MSOS, Typol appears to be less flexible with respect to the predicates that are allowed, though it is possible to switch between several inference systems. In addition, actions can be used in Typol to provide the rules with side effects, but we expect equivalent results to be expressive in MSOS labels in a more concise and elegant fashion. Finally, Typol seems, unfortunately, to be no longer available.

## 7 Conclusion

Developing static semantic analyzers for evolving languages requires the analysis techniques to co-evolve together with the syntax and the semantics of the language being analyzed. In particular, this means that the analyses should be flexible enough to accommodate different syntactical and semantical options contemplated by the language designers. In this paper we have shown that *modular structural operational semantics, introduced by Peter Mosses in [2,3], provides a well-suited framework for developing flexible semantic analyses*. To illustrate our approach we have chosen to formalize typing rules for expressions in Chi, a hybrid system specification language. We have seen that the analysis developed is indeed:

- flexible enough to co-evolve with the changing language (goal **(i)** from Section 1 addressed in Section 4.7),
- and that the MSOS transition rules can be implemented with a limited development effort (goal **(ii)** from Section 1 addressed in Section 5).

The MSOS-based approach advocated in this paper can be extended in a number of ways. First, the prototype implementation discussed in Section 5 can be further elaborated to provide for error reporting. In case MMT cannot rewrite an expression according to the MSOS transition rules, no specific action is taken.

Moreover, if desired, the Maude implementation can be used to generate the type checker code in a programming language chosen by the Chi 2.0 developers. To continue benefitting from the flexibility of the approach, automatic transformation techniques, e.g. [31], can be used.

Next, additional semantics-based analyses can be developed for Chi 2.0. One can, for instance, aim at detecting semantical errors in Chi 2.0 programs. Semantics of Chi 2.0 have been previously formalized using SOS [5], but the formalization is not well-suited for evolution. For example, it was found that adding concurrency or references to the functional language ML required a complete reformulation of the language specification [2]. Therefore, we consider reformulation of the Chi 2.0 semantics using MSOS or I-MSOS[32] as an important direction for future work. Expressing both the language semantics and the semantic analysis rules in the same semantic framework should lead to seamless integration of the two. Moreover, these translated rules can then be used as starting point to define the dynamic semantics in Action Semantics [33]. Such an Action Semantics definition of Chi2.0 allows another way of executing (and checking) the formal semantics of Chi2.0 via the action environment described in [34].

Finally, additional domain-specific languages should be considered.

## Acknowledgements

The authors are grateful to Albert Hofkamp, Michel Reniers and Ramon Schiffelers for reading and commenting on the early versions of this manuscript.

## References

1. Jürgens, E., Pizka, M.: Tool-supported multi-level language evolution. In: Männistö, T., Niemelä, E., Raatikainen, M. (eds.) Software and Services Variability Management Workshop – Concepts, Models and Tools, vol. (3), pp. 48–67. Helsinki University of Technology Software Business and Engineering Institute Research Reports, Helsinki (2007)
2. Mosses, P.D.: Modular structural operational semantics. *Journal of Logic and Algebraic Programming* 60-61, 195–228 (2004)
3. Mosses, P.D.: Foundations of modular SOS. In: Kutylowski, M., Wierzbicki, T., Pacholski, L. (eds.) MFCS 1999. LNCS, vol. 1672, pp. 70–80. Springer, Heidelberg (1999)
4. Hofkamp, A., Rooda, J.: Chi 1.0 reference manual. SE Report 2008-04, Eindhoven University of Technology, Systems Engineering Group, Department of Mechanical Engineering, Eindhoven, The Netherlands (2008)
5. van Beek, D., Hofkamp, A., Reniers, M., Rooda, J., Schiffelers, R.: Syntax and consistent equation semantics of Chi 2.0. SE Report 2008-01, Eindhoven University of Technology (2008)
6. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. *Theoretical Computer Science* 138, 3–34 (1995)

7. Bohdzieiewicz, J., Sroka, E.: Application of hybrid systems to the treatment of meat industry wastewater. *Desalination* 198(1-3), 33–40 (2006); The Second Membrane Science and Technology Conference of Visegrad Countries
8. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: The Maude 2.0 system. In: Nieuwenhuis, R. (ed.) RTA 2003. LNCS, vol. 2706, pp. 76–87. Springer, Heidelberg (2003)
9. Plotkin, G.D.: A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus (1981)
10. Braga, C., Haeusler, E., Meseguer, J., Mosses, P.D.: Mapping modular SOS to rewriting logic. In: Leuschel, M. (ed.) LOPSTR 2002. LNCS, vol. 2664, pp. 262–277. Springer, Heidelberg (2003)
11. Braga, C.: Rewriting Logic as a Semantic Framework for Modular Structural Operational Semantics. PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro (2001), <http://www.ic.uff.br/~cbraga>
12. Hofkamp, A., Rooda, J.: Chi 2.0 language reference manual. SE Report 2008-02, Eindhoven University of Technology, Systems Engineering Group, Department of Mechanical Engineering, Eindhoven, The Netherlands (2008)
13. Tofte, M.: Operational semantics and polymorphic type inference. PhD thesis, University of Edinburgh (1988)
14. Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge (2002)
15. Flanagan, C.: Hybrid type checking. In: Morrisett, J.G., Jones, S.L.P. (eds.) Principles of Programming Languages, pp. 245–256. ACM Press, New York (2006)
16. Mitchell, J.: Coercion and type inference. In: Eleventh Symposium on Principles of Programming Languages, pp. 175–185 (1984)
17. Hoang, M., Mitchell, J.C.: Lower bounds on type inference with subtypes. In: Proceedings of POPL 1995, 22nd Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 176–185 (1995)
18. Frey, A.: Satisfying systems of subtype inequalities in polynomial space. In: Van Hentenryck, P. (ed.) SAS 1997. LNCS, vol. 1302. Springer, Heidelberg (1997)
19. Meseguer, J., Rosu, G.: Rewriting logic semantics: From language specifications to formal analysis tools. In: Basin, D., Rusinowitch, M. (eds.) IJCAR 2004. LNCS (LNAI), vol. 3097, pp. 1–44. Springer, Heidelberg (2004)
20. Attali, I.: Compiling TYPOL with attribute grammars. In: Małuszyński, J., Deransart, P., Lorho, B. (eds.) PLILP 1988. LNCS, vol. 348, pp. 252–272. Springer, Heidelberg (1989)
21. Dijkstra, A., Swierstra, S.D.: Typing haskell with an attribute grammar. In: Vene, V., Uustalu, T. (eds.) AFP 2004. LNCS, vol. 3622, pp. 1–72. Springer, Heidelberg (2005)
22. Gao, J., Heimdahl, M.P.E., Van Wyk, E.: Flexible and extensible notations for modeling languages. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422, pp. 102–116. Springer, Heidelberg (2007)
23. Knuth, D.E.: Semantics of context-free languages. *Theory of Computing Systems* 2(2), 127–145 (1968)
24. de Moor, O., Backhouse, K., Swierstra, S.D.: First-class attribute grammars. *Informatica (Slovenia)* 24(3) (2000)
25. van Deursen, A.: An algebraic specification for the static semantics of Pascal. In: Conference Proceedings Computing Science in the Netherlands CSN 1991, pp. 150–164 (1991)
26. van Deursen, A., Heering, J., Klint, P. (eds.): Language Prototyping: An Algebraic Specification Approach. AMAST Series in Computing, vol. 5. World Scientific, Singapore (1996)

27. van den Brand, M., van Deursen, A., Heering, J., de Jong, H., de Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P., Scheerder, J., Vinju, J., Visser, E., Visser, J.: The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In: Wilhelm, R. (ed.) CC 2001. LNCS, vol. 2027, pp. 365–370. Springer, Heidelberg (2001)
28. Dinesh, T.B.: Type-checking revisited: Modular error-handling. In: Andrews, D.J., Groote, J.F., Middelburg, C.A. (eds.) Semantics of Specification Languages. Workshops in Computing, pp. 216–231. Springer, Heidelberg (1993)
29. Despeyroux, T.: Executable specification of static semantics. Research Report RR-0295, INRIA (1984)
30. Marcotty, M., Ledgard, H., Bochmann, G.V.: A sampler of formal definitions. ACM Computing Surveys 8(2), 191–276 (1976)
31. Lämmel, R., Wachsmuth, G.: Transformation of SDF syntax definitions in the ASF+SDF meta-environment. In: van den Brand, M.G., Parigot, D. (eds.) Proceedings of the 1st Workshop on Language Descriptions, Tools and Applications. Electronical Notes in Theoretical Computer Science, vol. 44, Elsevier, Amsterdam (2001)
32. Mosses, P.D., New, M.J.: Implicit propagation in structural operational semantics. In: SOS 2008, Preliminary Proceedings (2008)
33. Doh, K.G., Mosses, P.D.: Composing programming languages by combining action-semantics modules. Sci. Comput. Program. 47(1), 3–36 (2003)
34. van den Brand, M.G., Iversen, J., Mosses, P.D.: An action environment. Sci. Comput. Program. 61(3), 245–264 (2006)

# Action Algebras and Model Algebras in Denotational Semantics

Luiz Carlos Castro Guedes<sup>1</sup> and Edward Hermann Haeusler<sup>2</sup>

<sup>1</sup> Instituto de Computação, UFF, Niteroi, Brasil

<sup>2</sup> Departamento de Informática, PUC-Rio, Rio de Janeiro, Brasil

## 1 Introduction

This article describes some results concerning the conceptual separation of model dependent and language inherent aspects in a denotational semantics of a programming language. Before going into the technical explanation, the authors wish to relate a story that illustrates how correctly and precisely posed questions can influence the direction of research. By means of his questions, Professor Mosses aided the PhD research of one of the authors of this article and taught the other, who at the time was a novice supervisor, the real meaning of careful PhD supervision. The student's research had been partially developed towards the implementation of programming languages through denotational semantics specification, and the student had developed a prototype [12] that compared relatively well to some industrial compilers of the PASCAL language. During a visit to the BRICS lab in Aarhus, the student's supervisor gave Professor Mosses a draft of an article describing the prototype and its implementation experiments. The next day, Professor Mosses asked the supervisor, "Why is the generated code so efficient when compared to that generated by an industrial compiler?" and "You claim that the efficiency is simply a consequence of the Object- Orientation mechanisms used by the prototype programming language (C++); this should be better investigated. Pay more attention to the class of programs that might have this good comparison profile." As a result of these aptly chosen questions and comments, the student and supervisor made great strides in the subsequent research; the advice provided by Professor Mosses made them perceive that the code generated for certain semantic domains was efficient because it mapped to the "right aspect" of the language semantics. (Certain functional types, used to represent mappings such as Stores and Environments, were pushed to the level of the object language (as in `gcc`). This had the side-effect of generating code for arrays in the same way as that for functional denotational types. For example, PASCAL arrays belong to the "language inherent" aspect, while the Store domain seems to belong to the "model dependent" aspect. This distinction was important because it focussed attention on optimizing the model dependent semantic domains to obtain a more efficient implementation.) The research led to a nice conclusion: The guidelines of Action Semantics induce a clear separation of the model and language inherent aspects of a language's semantics. A good implementation of facets, particularly the model dependent ones, leads to generation of an efficient compiler. In this article we discuss the separation of the

language inherent and model-inherent domains at the theoretical and conceptual level. In doing so, the authors hope to show how Professor Mosses's influence extended beyond his technical advice to his professional and personal examples on the supervision of PhD research.

## 2 Motivation and Statement of Purpose

Many authors[1,6,11,12] have said that Mosses observed that denotational semantics intertwines model details with the underlying conceptual analysis. However, it seems that they did not have try to precisely define what the words "model details" and "underlying conceptual analysis" really mean. On the other hand, Mosses suggested that the arbitrary choice of semantic domains at the denotational semantics descriptions may be harmful to the quality of the compilers automatically generated from those descriptions. Thus, he has proposed the abstract semantic algebra approach[8] in order to avoid that choice and therefore ensure acceptable quality standards for the automatically generated compilers. This approach lead to the Action Semantics framework for programming language specification. The concepts of facets and actions provide, for well-known language styles, a nice sharing between model details and conceptual analysis. However, for nonconventional languages, the designer may have to define new unforseen facets and it can be argued wether the necessary amount of work in doing that is comparable to the effort required to write (and actually read) a traditional denotational semantics specification. Guedes[3] has shown that an object oriented approach to compiler generation give raise to a natural separation of such concepts. One of the main proposals of this paper is to formalize the algebraic concepts underlying such natural separation, looking forward to a language dependent, instead of specification dependent, definition of being inherent to a language. In this paper we will precisely define the words "model details" and "underlying conceptual analysis" by means of a semantic domain separation. We will partition the semantic domains in two classes: a class for the model dependent domain and a class for the language inherent domains and will show how to find which class a domain belongs to. We propose the concept of model algebra in order to specify the domain separation and the concept of action algebra in order to specify the reduction of a denotational semantics description to an equivalent action semantics description. The action algebra is the  $S$ -sorted  $\Sigma$  - -algebra  $\mathcal{M}$ , where  $S$  is the set of model dependent semantic domains,  $\Sigma$  is a  $S$ -sorted signature built with the elements of the language inherent semantic domains and  $E$  is the set of  $\Sigma$ -equations defined by the semantic equations of the denotational semantics description that satisfies  $\mathcal{M}$ . The action algebra of a denotational semantics description corresponds to the target algebra of the equivalent action semantics description. This results shows that the proposed separation is in agreement with the separation performed by Action semantics. Looking towards a definition of what should be the best semantic description of a programming language we have evolved the concept of Action algebra. Considering the Action algebras of all Denotational Semantics description of a language

$\mathcal{L}$ , built over the same abstract syntax, together with the  $\Sigma$ -homomorphisms between them, we have a category of Action algebras, where the initial algebra is named here as the Language Algebra. The Language algebra is by construction the smallest algebra that completely defines  $\mathcal{L}$ . Thus, it may be considered that it came from the best specification of  $\mathcal{L}$ .

In [15] it is develop a two-level denotational semantics aiming to differentiate between run-time and compile-time features of a language specification. This distinction between the two levels is taken to the extreme, by providing different syntax for compile-time and run-time expressions: lambda-terms for the former, combinators for the latter. This approach has the same motivation of ours, since it focus on a distinction between computation model and language model. In its framework for compiler construction, the generated code optimization is mainly focused on the optimization of the run-time features. Our result, on the other hand, is focused on answering how one can adequately identify these two-levels in a language. It is worth to mention that our answer to this question is in agreement with the fact that Action Semantics guides the specifier to a good separation between the computation model and language inherent model. This fact was perceived during the research reported in this work.

### 3 Basic Definitions and Background

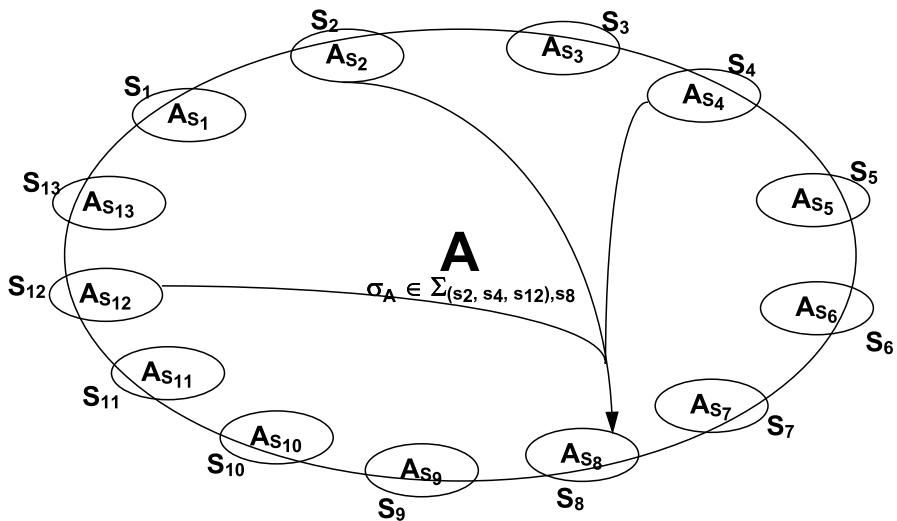
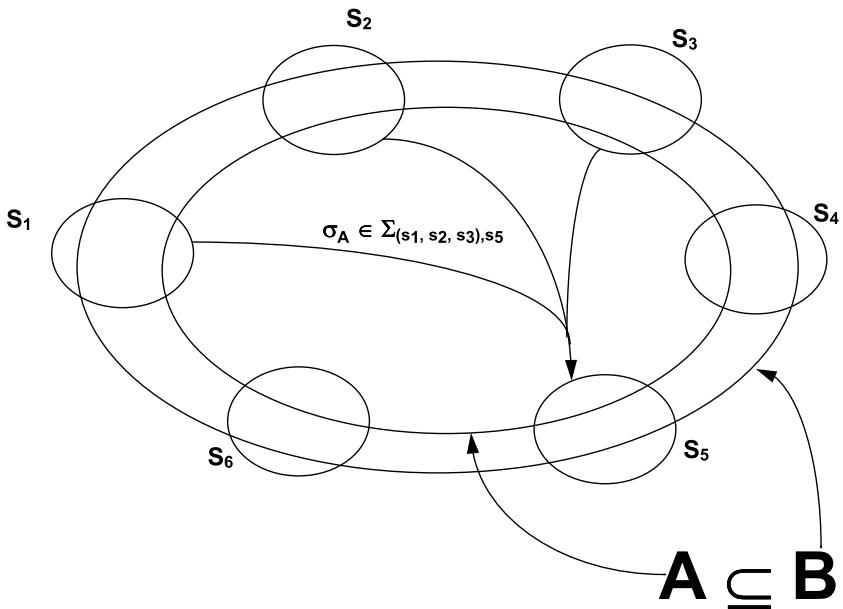
In this section we remind the reader of the notation and terminology employed in this article. We will follow the notation of the ADJ group[4] and extend some concepts of abstract data type description in order to conform programming language descriptions, as developed in[8]. The proof of the propositions and theorems of this section may be found primarily in [4]. It is worth mention that there are books that are quite helpful in teaching these notions, as [5,2].

**Definition 1.** *For any set  $S$ , an  $S$ -sorted signature  $\Sigma$  is a  $(S^* \times S)$ -indexed family of sets of operators,  $\Sigma(s_1, \dots, s_n), s$ . The index  $(s_1, \dots, s_n)$  denotes the arity of the operators and the index  $s$  is the sort of result,  $s_1, \dots, s_n, s \in S$ .*

**Definition 2.** *Let  $\Sigma$  be a  $S$ -sorted signature. A  $\Sigma$ -algebra  $\mathcal{A}$  is a  $S$ -indexed family of sets  $A_{S_i}$ ,  $s_i \in S$ , together with a finite set of total functions  $\sigma_A : (A_{s_1} \times \dots \times A_{s_n}) \rightarrow A_s$ , one for each  $\sigma \in \Sigma(s_1, \dots, s_n), s$ .  $A_s$  is called the carrier of  $\mathcal{A}$  with sort  $s$ , and  $\sigma_A$  is an operation on  $\mathcal{A}$  named by  $\sigma$ , or it is a element of  $A_s$ , when  $\sigma \in \Sigma(), s$ . If  $S$  has just one sort,  $\mathcal{A}$  is called a single-sorted algebra, otherwise it is called a many-sorted algebra.*

In this article we depict a  $\Sigma$ -Algebra  $\mathcal{A}$  and its operators as in figure 1. In this figure take  $A_{S_i}$  is being completely inside  $A$ . The same observation holds for figure 2. The carriers of  $\mathcal{A}$  correspond to the inside portion of each sort.

**Definition 3.** *Let  $\mathcal{A}$  and  $\mathcal{B}$  be two  $S$ -sorted  $\Sigma$ -algebras. We say that  $\mathcal{A}$  is a sub-algebra of  $\mathcal{B}$ ,  $\mathcal{A} \subseteq \mathcal{B}$ , iff, all of  $\mathcal{A}$ 's carriers,  $A_{s_i}$ , are contained inside the*

Fig. 1. A  $\Sigma$ -Algebra schemeFig. 2. Two  $\Sigma$ -Algebras  $A \subseteq B$ 

corresponding carriers of  $\mathcal{B}$ ,  $B_{s_i}$ . More precisely, for every  $i$ ,  $A_{s_i} \subseteq B_{s_i}$ , and for each operation  $\sigma \in \Sigma$ ,  $\sigma_A(a_1, \dots, a_n) = \sigma_B(a_1, \dots, a_n)$ , for all  $a_i \in A_{s_i}$ .

Figure 2 depicts  $\mathcal{A} \subseteq \mathcal{B}$ .

**Definition 4.** Let  $\mathcal{A}$  and  $\mathcal{B}$  be two  $S$ -sorted  $\Sigma$ -algebras. A  $\Sigma$ -homomorphism  $h : \mathcal{A} \rightarrow \mathcal{B}$  is an  $S$ -indexed family of  $h_s : A_s \rightarrow B_s$ , such that  $h_s(\sigma_A(a_1, \dots, a_n)) = \sigma_B(h_{s_1}(a_1), \dots, h_{s_n}(a_n))$ , for each  $\sigma \in \Sigma(s_1, \dots, s_n)$ ,  $s$ , and,  $a_i \in A_{s_i}$ . If  $\sigma \in \Sigma()$ ,  $s$  then  $h_s(\sigma_A) = \sigma_B$ .

Figure 3 shows a homomorphism.

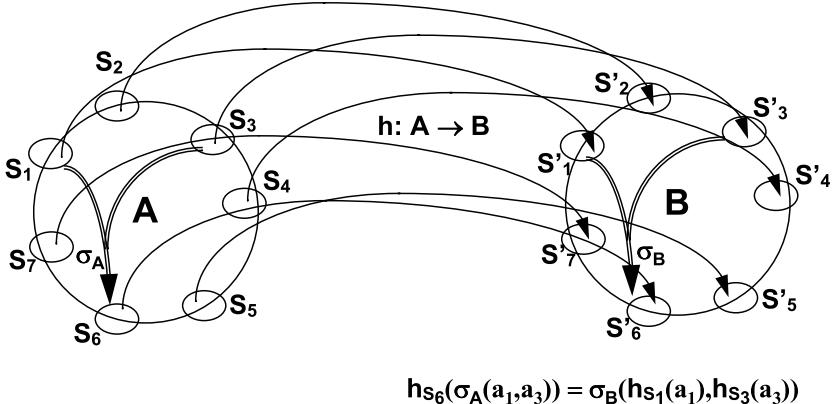


Fig. 3.  $\Sigma$ -Homomorphism Scheme

**Definition 5.** A set  $C$  of  $S$ -sorted  $\Sigma$ -algebras, together with all  $\Sigma$ -homomorphisms between them is called a category of  $\Sigma$ -algebras. The set of algebras that forms  $C$  is called  $Obj(C)$  and the set of homomorphisms is called  $Mor(C)$ .

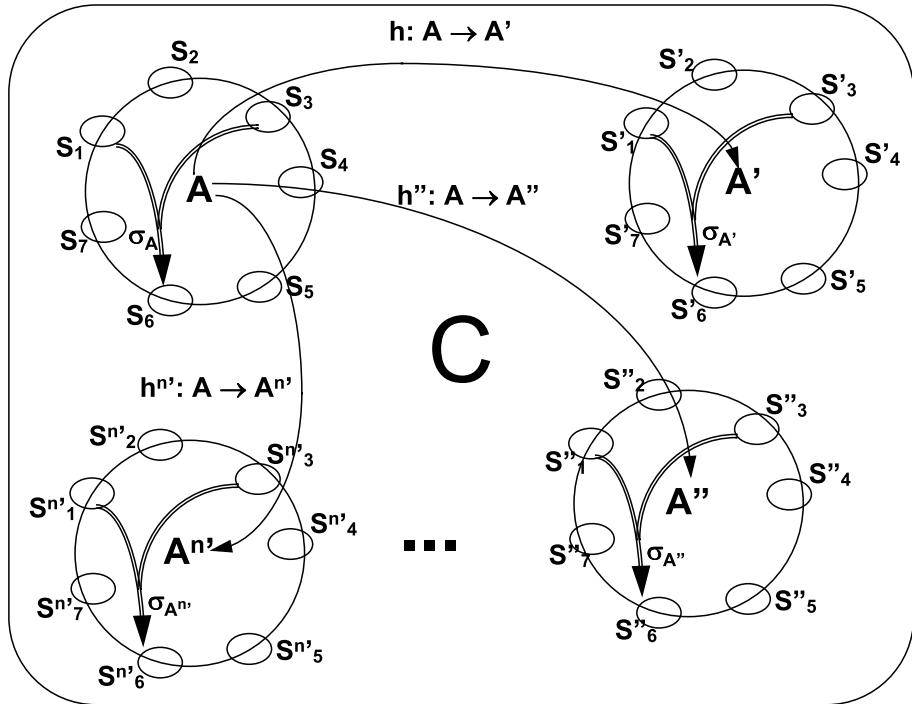
**Definition 6.** Given a  $S$ -sorted  $\Sigma$ -algebra  $\mathcal{A}$ , the identity  $\Sigma$ -homomorphism over  $\mathcal{A}$ , noted by  $I_{\mathcal{A}}$ , is built by the  $S$ -indexed family of identity functions over the carriers  $A_s$ .

**Definition 7.** Let  $\mathcal{A}$  and  $\mathcal{B}$  be two  $S$ -sorted  $\Sigma$ -algebras of a category  $C$ . We say that  $\mathcal{A}$  and  $\mathcal{B}$  are isomorphic as  $\Sigma$ -algebras, iff, they are isomorphic as objects of the category. The  $\Sigma$ -homomorphism that shows that they are isomorphic is also called a  $\Sigma$ -isomorphism.

**Definition 8.** A  $\Sigma$ -algebra  $\mathcal{A}$  is called initial in a category  $C$  of  $\Sigma$ -algebras, if and only if, for every  $\Sigma$ -algebra  $\mathcal{B}$  in  $C$ , there exists a unique homomorphism  $h : \mathcal{A} \rightarrow \mathcal{B}$

Figure 4 shows a category  $C$  and the homomorphisms from its initial algebra to other algebras.

The initial algebra is in fact an initial object of the category (see [2]). Thus, there is only one initial algebra, modulo isomorphisms ( $\Sigma$ -homomorphisms). The fact that there must be an initial algebra, for every category of  $\Sigma$ -Algebras, comes by observing that the initial algebra is nothing more than the Algebra of terms on the signature  $\Sigma$ . We denote by  $Alg_{\Sigma}$  the category of all  $\Sigma$ -Algebras, for a signature  $\Sigma$ .



**Fig. 4.** A Category of  $\Sigma$ -Algebras and its Initial object

When considering equations we evolve to the concept of  $\Sigma, E$ -Algebras, where  $E$  is a set of  $\Sigma$ -sorted equations. In this case, the equations induce a congruence relation on each carrier  $A_s$  of a  $\Sigma, E$ -Algebra  $\mathcal{A}$ . In this case the quotient algebra is taken as the initial object of the category. We denote  $Alg_{\Sigma, E}$  the category of all  $\Sigma, E$ -Algebras and  $\Sigma$ -homomorphisms.

#### 4 Model Algebras and Action Algebras

In this section, we show the fundamental concepts to the separation between model dependent domains and language inherent domains. The general idea behind this separation is that the model dependent domains must, directly or indirectly, contribute to the specification of the language inherent ones. A result that we intend to establish here is that the language inherent domains must have a correspondent domain in all descriptions of the same language, despite of the chosen model (the model dependent domains). We use the denotational semantics description of an imperative language  $\mathcal{L}$  to illustrate the concepts presented through out this paper. It is shown in figure 5.

**Definition 9.** We define *Signs* to be the many-sorted signature defined by the abstract syntax of  $\mathcal{L}$  in  $S$ , according to [8], whenever a denotational semantics description  $S$  of a language  $\mathcal{L}$  is provided.

Syntactic Domains:	Semantic Domains:
$\Pi$ : Prog (Programs)	$n$ : Names ;
$\Delta$ : Decl (Declarations)	$i$ : Integers ;
$\Sigma$ : Stmt (Statements)	$l$ : Locations ;
$E$ : Expr (Expressions)	$s$ : Stores = Locations $\rightarrow$ Integers ;
$I$ : Ide (Identifiers)	$p$ : Procedures = Locations $\rightarrow$ Stores $\rightarrow$ Stores ;
$N$ : Num (Numbers)	$v$ : Values = Locations + Procedures ;
Abstract Syntax:	$e$ : Environments = Names $\rightarrow$ Values ;
$\Pi = \Delta ; \Sigma$	Semantic Functions:
$\Delta = I$	$P$ : Prog $\rightarrow$ Stores $\rightarrow$ Stores ;
$  I_1 (' I_2 )' \Leftarrow \Sigma$	$D$ : Decl $\rightarrow$ Environments $\times$ Stores $\rightarrow$ Environments $\times$ Stores ;
$  \Delta_1 ; \Delta_2$	$S$ : Stmt $\rightarrow$ Environments $\rightarrow$ Stores $\rightarrow$ Stores ;
$\Sigma = I \Leftarrow E$	$E$ : Expr $\rightarrow$ Environments $\rightarrow$ Stores $\rightarrow$ Integers ;
$  I_1 (' I_2 )'$	$I$ : Ide $\rightarrow$ Name ;
$  \Sigma_1 ; \Sigma_2$	$N$ : Num $\rightarrow$ Integers ;
$  (' \Delta ; \Sigma_1 )'$	
$E = E_1 ; E_2$	
$  N$	
$  I$	
$I = \text{token} ;$	
$N = \text{token} ;$	
Semantic Equations:	
$P[[\Delta ; \Sigma]](s) = S[[\Sigma]](D[[\Delta]](e,s), s), e = \lambda n. \perp;$	
$D[[I]](e,s) = < e(I[[I]] \leftarrow l), s(l \leftarrow \perp) >, l = \text{free}(s) ;$	
$D[[\Delta_1 ; \Delta_2]](e,s) = (D[[\Delta_1]] \circ D[[\Delta_2]])(e,s) ;$	
$D[[I_1 (' I_2 )' \Leftarrow \Sigma]](e,s) = < e(I[[I]] \leftarrow p), s >, p(l) = S[[\Sigma]](e(I[[I_2]] \leftarrow l)) ;$	
$S[[I \Leftarrow E]](e,s) = s(e(I[[I]] \leftarrow E)(e,s)) ;$	
$S[[I_1 (' I_2 )']] (e,s) = p(l,s), l = e(I[[I_2]]), p = e(I[[I_1]]) ;$	
$S[[\Sigma_1 ; \Sigma_2]](e,s) = (S[[\Sigma_1]](e) \circ S[[\Sigma_2]](e))(s) ;$	
$S[[(' \Delta ; \Sigma_1 )']] (e,s) = S[[\Sigma_1]](e, s_1), < e_1, s_1 > = D[[\Delta]](e,s) ;$	
$E[[E_1 ; E_2]](e,s) = E[[E_1]](e,s) + E[[E_2]](e,s) ;$	
$E[[N]](e,s) = N[[N]] ;$	
$E[[I]](e,s) = s(e(I[[I]])) ;$	
$I[[I]] = \text{token to Names} ;$	
$N[[N]] = \text{token to Integers} ;$	

Fig. 5. Denotational Description of  $\mathcal{L}$ 

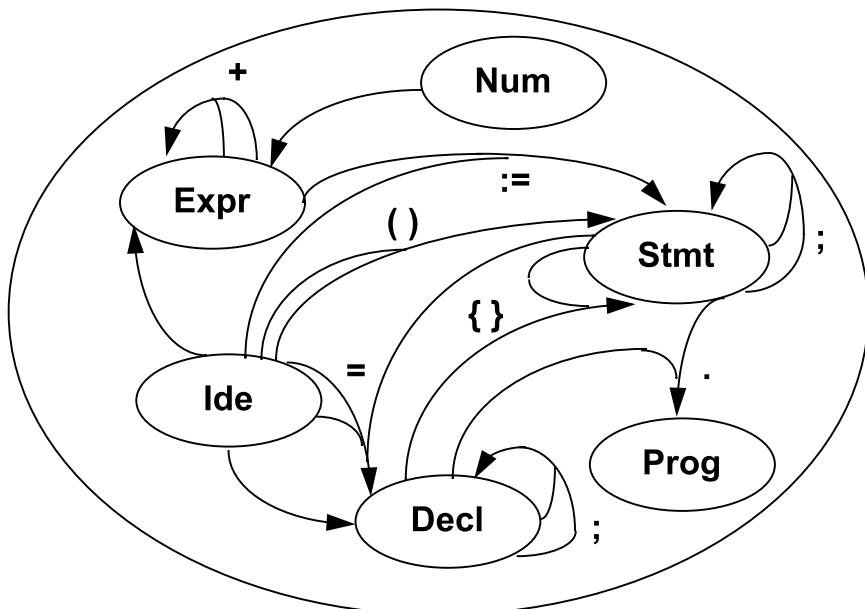
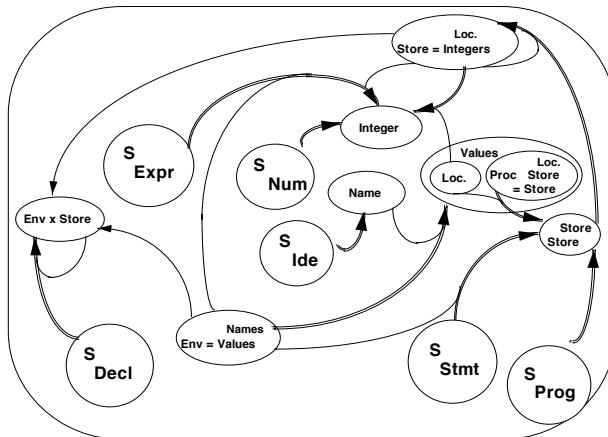
In the following definitions  $S$  is to be taken as a denotational semantic description of a programming language  $\mathcal{L}$ .

We denote by  $Init_S$  the initial algebra of the category  $Alg_{\Sigma,E}$  of all many sorted algebras with signature  $\Sigma = Signs_S$ , and that satisfies the set of semantic equations  $E$  of  $S$ . Figure 6 illustrates the initial algebra signature of the example language description.

The semantic algebra built with all semantic domains and the operations over them, usually do not have the same signature as the syntactical algebra  $Init_S$ . Figure 7 illustrates this situation regarding our denotational specification example. The double lined arrows correspond to the evaluation morphisms of the functional domains.

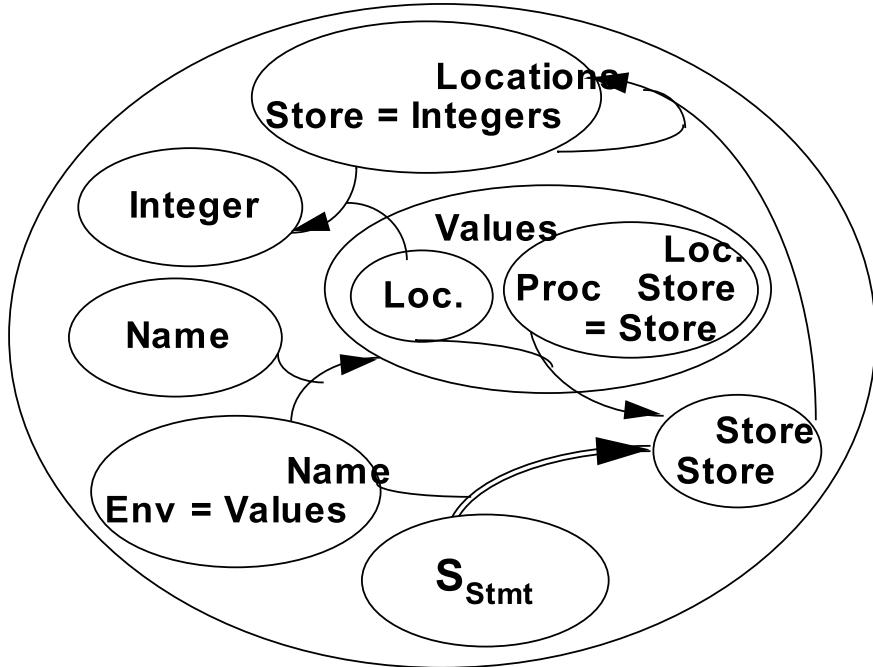
As told before, we need that both syntactic and semantic algebras have the same signature in order to allow the semantic functions to be considered homomorphisms. We also need a kind of representation of the semantic algebra that might help in finding the separation between the model dependent domains and the language inherent ones.

The following definitions aim to show how to gather the semantic domains of the target algebra into self-contained sub-algebras so that we obtain a new algebra with same signature of the syntactical algebra.

Fig. 6. Initial Algebra of  $\mathcal{L}$ Fig. 7. Target Algebra signature for the denotational spec of  $\mathcal{L}$ 

**Definition 10.** We define the domain algebra of  $D$  to be the semantic algebra having all the semantic domains involved in the definition of  $D$  as its sorts and a family of evaluation morphisms over those sorts as its signature.

Figure 8 illustrates the domain algebra signature of the denotation of the syntactic domain  $Stmt$ ,  $S_{Stmt}$ , of the example language. The double lined arrow represent the evaluation morphism of the semantic domain that it corresponds to.



**Fig. 8.** The Domain Algebra of *Stmt*

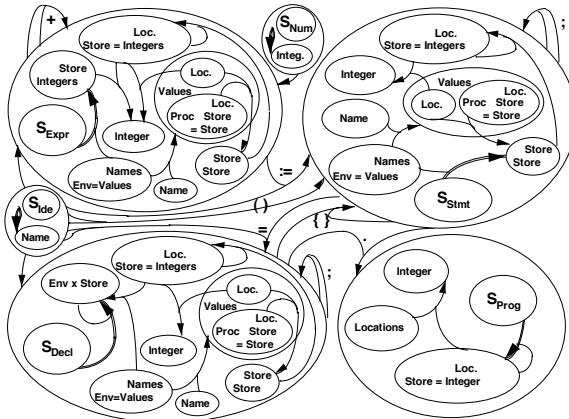
**Definition 11.** We say that a semantic domain  $D$  is a denotational domain if  $D$  reflects the denotation of a syntactic domain phrase. The domain algebra of a denotational domain is called a denotational algebra.

Figure 8 shows a denotational algebra.

**Definition 12.** We define the target algebra of  $S$ , namely  $Targs$ , to be the semantic algebra having the denotational algebra as the carriers of the syntactic sorts, satisfying the set of semantic equations  $E$  on  $S$ ,  $Targs \in Obj(Alg_{\Sigma},)$ .

Figure 9 illustrates a target algebra of the example language specification. Note that the hidden elements of this algebra mainly carry model (computational) dependent information needed to assign semantics to the language inherent aspects of the specified language. It is worth noting that we can take the carrier set of a  $Targs$  sort as an algebra itself. We have only to take the hidden elements into account. This discussion resembles, partially, the motivation for the definition of Hidden Algebras[13].

An extension of a  $\Sigma$ -algebra  $A$  is a  $\Sigma'$ -algebra  $B$ , such that, there is a  $\Sigma$ -homomorphism that is a, set-theoretically, inclusions from the carriers in  $A$  into their corresponding in  $B$ . Note that  $\Sigma'$  may extend  $\Sigma$ . We also say that  $A$  is a sub-algebra of  $B$ , restricted to the signature of  $A$ , or equivalently that  $B$  is a super-algebra of  $A$ . In what follows,  $\prec$  stands for the algebra extension/restriction relationship between algebras.



**Fig. 9.** Target algebra signature and intermediate algebras

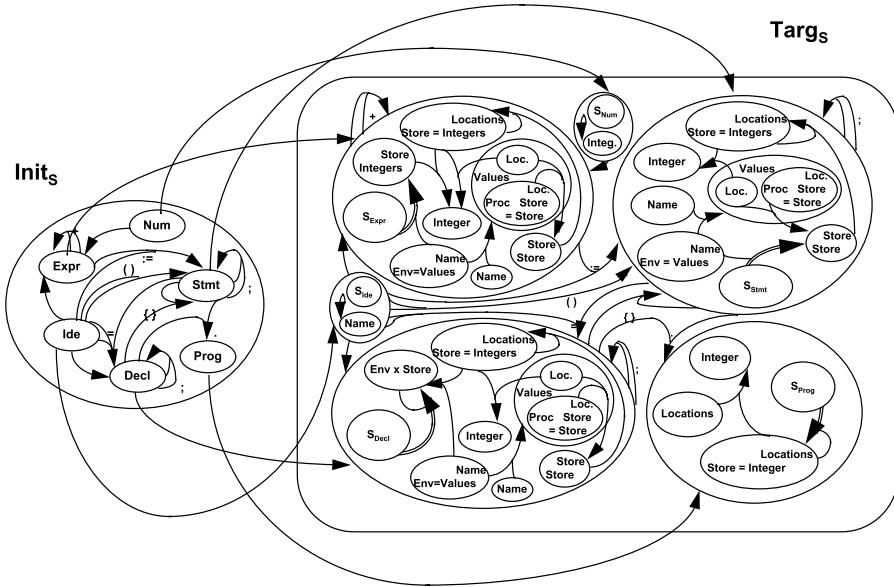
**Definition 13.**  $\tau$  is an intermediate algebra of  $Targs$  iff for each carrier  $\beta$  in  $Targs$ ,  $\tau \prec \beta$ , or  $\beta \prec \tau$ , where  $\tau$  and beta are taken as algebras.

In figure 9, the carrier algebras for *Location*, *Store* and *Integer* are examples of intermediate algebras. On the other hand, neither *Values* nor *Name* are not intermediate algebras in  $Targs$ . The reader can see that  $\prec$  is a partial ordering relation on algebras. Thus, given a set  $X$  of algebras, an algebra  $A$  is greater than  $X$ , whenever for each  $B \in X$ ,  $B \prec A$ . A greatest algebra in  $X$ , if exists, is any algebra  $A \in X$  that is greater than  $X$ , and, such that, any other algebra  $C \in X$ , greater than  $X$  is  $A$  itself. Note that this terminology is not the usual one, that considers isomorphisms. The reason for that is clear from the observations in the following paragraph.

The reader can note that the greatest intermediate algebra, in this example (figure 9), carries all information about stores and its relationship to values. One can take this algebra as representing model dependent domains of the specified language. It involves *Location*, *Store*, *Integer* and *Values* as well the evaluation morphisms and specific operations inner to each hidden element algebra. The reader can also note that given a target algebra of a denotational semantics specification of a language, the greatest intermediate algebra, if exists, is unique. This is a consequence of the definition of  $\prec$  being in terms of inclusions, instead of mono-morphisms.

It is interesting to notice that  $Init_S$  has the same signature of  $Targs$ , and, obviously, the set of semantic functions of  $S$  builds a homomorphism,  $H$ , from  $Init_S$  to  $Targs$ . Figure 10 shows the homomorphism between them.

**Definition 14.** Given a denotational semantics description  $S$  of a language  $\mathcal{L}$ , we say that a semantic domain  $D$  is a scalar domain if  $D$  is a primitive semantic domain or the coproduct of scalar domains.



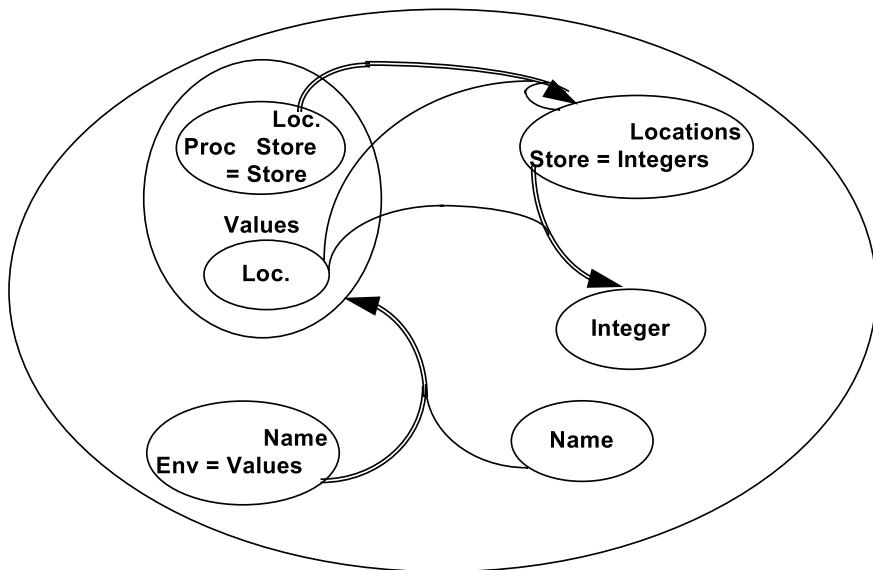
**Fig. 10.** Semantic homomorphism for the language  $\mathcal{L}$

**Definition 15.** Given a denotational semantics description  $S$  of a language  $\mathcal{L}$  and two semantic domains  $D$  and  $D'$  of  $S$ , we say that  $D$  is compatible with  $D'$ , noted as  $D \approx D'$ , if one of the following conditions is satisfied:

- if  $D = D_1 + \dots + D_n$  and  $D' = D'_i$ ,  $1 \leq i \leq n$ , then  $D \approx D'$ .
- if  $D = D_1 + \dots + D_n$  and  $D' = D'_1 + \dots + D'_n$  and, for each  $i$ ,  $1 \leq i \leq n$ ,  $D_i \approx D'_i$ , then  $D \approx D'$ .
- if  $D = D_1 \times \dots \times D_n$  and  $D' = D'_1 \times \dots \times D'_n$ , and, for each  $i$ ,  $1 \leq i \leq n$ ,  $D_i \approx D'_i$ , then  $D \approx D'$ .
- if  $D = D_1 \rightarrow D_2$  and  $D' = D'_1 \rightarrow D'_2$ ,  $D_1 \approx D'_1$  and  $D_2 \approx D'_2$ , then  $D \approx D'$ .
- if  $D = D_1^*$  and  $D' = (D'_1)^*$  and  $D_1 \approx D'_1$  then  $D \approx D'$ .
- if  $D$  and  $D'$  are primitive domains and  $D = D'$  then  $D \approx D'$ .

This definition states that, besides the usual structural compatibility, a coproduct domain is compatible with its component domains. This happens to guarantee that functional domains will not lose their characteristic of language inheritance when they are defined as integrants of a coproduct that is the denotation of a syntactical phrase, as we can see at the next definition, one of the motivations of this article. The reader can also note that  $\approx$  is an equivalence relation on the semantic domains of a language specification.

**Definition 16.** Given a denotational semantics description  $S$  of a language  $\mathcal{L}$ , we say that a semantic domain  $D$  is model dependent if  $D$  is a sort of the greatest intermediate algebra of  $Targs$  and  $D$  is not a functional domain compatible with any denotational domain. Semantic domains that are not model dependent are called language inherent.



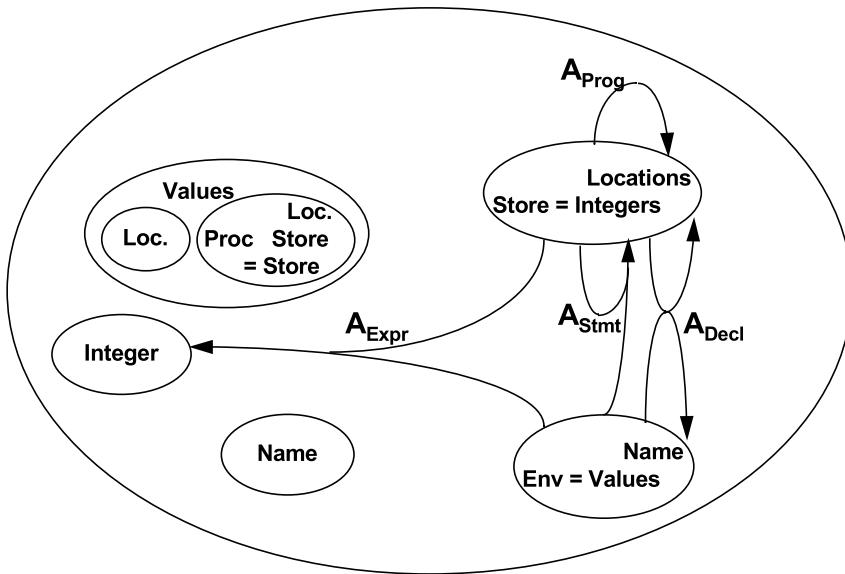
**Fig. 11.** Model Algebra of the example language

This definition states that the functional denotations of syntactical phrases and some named semantic domains are language inherent. The model dependent domains are the ones that exist to build the language inherent domains. For instance, given a denotational description of an imperative language  $\mathcal{L}$ , the denotation of the statements may be a semantic domain with a structure like  $D$  for kind  $A \rightarrow (B \rightarrow B)$ . This structure of  $D$  is inherent to the statement semantics in  $\mathcal{L}$ , no matter what the domains  $A$  and  $B$  look like, the model of  $D$ . It is interesting to observe that a semantic domain  $D$  may fall into different classes depending on the language it occurs. Although, for a given language it should always fall into the same class.

**Definition 17.** *Given a denotational semantics description  $S$  of a language  $L$ , we define the model algebra of  $S$  to be the sub-algebra of  $Targs$  whose set of sorts is built with the model dependent domains and signature operators are given by their evaluation morphisms.*

The idea that has suggested the model algebra definition is that it should be involved with every sub-algebras of the target algebra, that is, any other sub-algebra of  $Targs$  is built with the model algebra or is used to build it. Figure 11 illustrates the model algebra signature of the example language specification.

**Definition 18.** *Given a denotational semantics description  $S$  of a language  $\mathcal{L}$ , we define  $Acts_S$ , the action algebra of  $S$ , to be the many sorted algebra which set of sorts is the same of the model algebra and signature operators are given by the elements of language inherent domains that occurs in  $S$ .*



**Fig. 12.** Action Algebra of the example language (I)

The action algebra may be considered as a new version of the original target algebra with the set of sorts restricted to the sorts of the model algebra and having the elements of the other domains (language inherent) as the operators of the signature. If we represent the operations of the action algebra with letter  $A$ , of action, indexed with the name of the domains they give meaning to, we will have the scheme shown at figure 12.

In order to avoid the problems brought by a bad choice of semantic domains when describing a programming language, we have suggested the separation between model dependent domains and language inherent domains. The idea is that it is possible to enhance the implementation of a language processor just improving the quality of the implementation of the model dependent domains. With this we can solve the same efficiency problem that action semantics has solved.

Next section we will show how the separation proposed here may be used to obtain an action semantics description from the matching of the facets with an equivalence class of the sorts of the model algebra. This result reinforces the separation proposed here and explains why it is possible to automatically generate efficient compilers from denotational semantics. The reduction of a denotational semantics description to a formalism (action semantics) that, presumably, already makes the this separation ensures the legitimacy of the separation proposed here.

## 5 Denotational Semantics Separability

The action algebra of the denotational description of a language  $\mathcal{L}$  shows a close resemblance with the target algebra of an action semantics description of  $\mathcal{L}$ . If

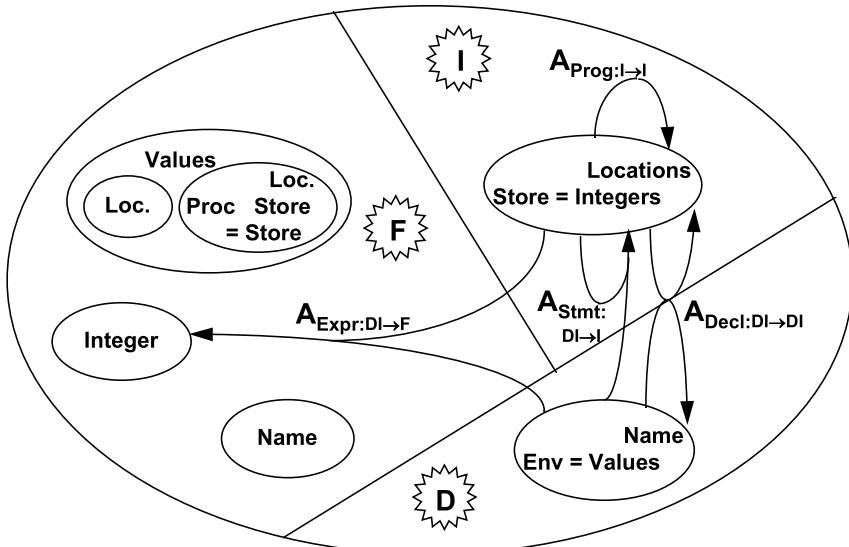
we were able to find out a partition of the set of sorts of the action algebra of  $\mathcal{L}$  which each equivalence class could be matched with a facet of the action semantics of  $\mathcal{L}$ , then we would easily build an action semantics description of  $\mathcal{L}$  by redefining the operators of the action algebra to act over the classes of the corresponding domains and translating their  $\lambda$ -term to an equivalent action.

At this point, if we match, for the example language, the set built with the scalar domains and the Procedure domain with the functional facet, the set built with the Store domain with the imperative facet and the set built with the Environment domain with the declarative facet, naming them by  $F$ ,  $I$  and  $D$ , respectively, we will obtain an algebra that is closely like the target algebra of an action semantics of the example language, as shown in figure 13.

**Theorem 1.** *Given a denotational semantics description  $S$  of a language  $\mathcal{L}$ , the action algebra of  $S$  implicitly defines an action semantics description  $A$  of  $\mathcal{L}$ .*

**Proof.** Sketch (from [3]): It is enough to show how to build an Action Semantics Description,  $A$ , for  $\mathcal{L}$  from  $S$  and prove that this building preserves the semantics. The abstract syntax of  $\mathcal{L}$  remains the same. The semantic functions have to be rewritten, as shown below. The action algebra of  $S$  induces a partition of the semantics domains into the well-known facets of Action Semantics ( $F$ ,  $D$ ,  $I$ ,  $C$  and  $U$ ). We use the notation  $X(D)$ , where  $D$  is a domain from  $S$  to denote facet to which  $D$  belongs to, considering the action algebra of  $S$ .  $\prod(C_1, \dots, C_n)$  denotes  $C_1 C_2 \dots C_n$ , where  $C_i$  each  $C_i$  is a facet. Thus, any semantic function

$$F : Dom \rightarrow D_1 \rightarrow \dots \rightarrow D_n \rightarrow D$$



**Fig. 13.** Action Algebra of the example language (II)

becomes

$$A_F :: Dom \rightarrow Action_{F_1 \rightarrow F_2}$$

where

$$F_1 = \prod_{i=1}^n (\bigcup X(D_i))$$

and

$$F_2 = X(D)$$

Any semantic equation  $F[[\Delta]] = M$  becomes  $F[[\Delta]] = A$ . Figure 14 defines  $A$  according the rewriting rules inside the boxes. We also assume that  $A_i : K_i \rightarrow K'_i$ ,  $\kappa_i \in K_i$ ,  $\kappa \in K$ ,  $\rho \in D$ ,  $\sigma \in I$ ,  $\phi \in F$  and that  $()$  represents the basic facet. At the right-hand side of the symbol ‘:’ we have the type of actions. Following the usual terminology of Action Semantics,  $F$  is the functional facet,  $I$  is the imperative facet and so on.

The proof of the semantics invariance, of the S-to-A translation, is by induction on the complexity of the  $\lambda$ -terms occurring at the right-hand side of each semantic equation of  $S$ . The basis of the induction is the proof that the translation preserves semantics for the most elementary  $\lambda$ -terms, namely a single variable, a constant value, the undefined value and the error. The axioms at the top of the box at Figure 14 ensure the correctness of these translations. The induction step is based upon that the translation is correct (semantic preserving) for each  $\lambda$ -term  $M_i$  and it must be proven that, for every  $\lambda$ -term  $M$  built with the  $M_i$ 's, the translation is also correct. We remember that Figure 14 shows how  $M$  is translated depending on its outmost operator. The soundness of each translation rule must be proven. We only show a couple of them. Actions' meaning are taken from figures 3 and 4 of [7]. We alert the reader that figure 14 does not contain all cases in the definition of the rewriting rules. Lack of space prevent us to include the whole picture. The reader is referred to [3] to see the whole set of rewriting rules.

- The translation of  $M \equiv M_1 \star M_2$  is sound. Consider  $M_1 \star M_2$  in its raw form, that is,  $\lambda\rho\lambda\sigma.M_2(\rho, M_1(\rho, \sigma))$ , and, the “ $A_1$  and then  $A_2$  combinator semantics given by  $\lambda\rho\lambda\sigma.A_2(\rho, A_1(\rho, \sigma))$ . Then the soundness follows directly from the inductive hypothesis concerning  $M_i$  and  $A_i$ .
- The translation of  $M \equiv M_1 M_2$  is sound. Its raw form<sup>1</sup> is  $\lambda\kappa_1\lambda\kappa_2.M_1(\kappa_1)(M_2(\kappa_2))$ . Considering the semantics of the combinators:

$$\begin{array}{ll} A_1 \text{ and } A_2 \equiv \lambda\kappa_1\lambda\kappa_2.\langle A_1(\kappa_1), A_2(\kappa_2) \rangle, & A_1 \text{ then } A_2 \equiv \lambda\kappa.A_2(A_1(\kappa)) \\ \text{call} \equiv \lambda\phi\lambda\pi\lambda\sigma.\phi(\pi, \sigma), & \text{apply} \equiv \lambda\phi\lambda\pi.\phi(\pi) \end{array}$$

Thus, we have following possible cases, where the last belongs, in fact, to the basis of induction.

---

<sup>1</sup> In order to improve readability, we use the same names for binded variables in the raw form of a  $\lambda$ -term than the names of variables in the context of the corresponding rewriting rule in figure14.

$$\vdash I \approx I : () \rightarrow ()$$

$$\vdash n \approx \text{give } n : () \rightarrow F$$

$$\vdash \perp \approx \text{diverge} : () \rightarrow ()$$

$$\vdash \top \approx \text{fail} : () \rightarrow F$$

$$\frac{\rho, \sigma \vdash M_1 \approx A_1 : DI \rightarrow I \quad \rho, \sigma' \vdash M_2 \approx A_2 : DI \rightarrow I}{\rho, \sigma \vdash M_1 * M_2 \approx A_1 \text{ and then } A_2 : DI \rightarrow I}$$

$$\frac{\kappa_1 \vdash M_1 \approx A_1 : K_1 \rightarrow F \quad \kappa_2 \vdash M_2 \approx A_2 : K_2 \rightarrow F}{< \kappa_1, \kappa_2 > \vdash < M_1, M_2 > \approx A_1 \text{ and } A_2 : K_1 \cup K_2 \rightarrow F}$$

$$\frac{\kappa_1 \vdash M_1 \approx A_1 : K_1 \rightarrow K_2 \quad \kappa_2 \vdash M_2 \approx A_2 : K_2 \rightarrow K'_2}{\kappa_1 \vdash M_1 \circ M_2 \approx A_1 \text{ thence } A_2 : K_1 \rightarrow K'_2}$$

$$\frac{\kappa_1 \vdash M_1 \approx A_1 : K_1 \rightarrow F \quad \kappa_2 \vdash M_2 \approx A_2 : K_1 \rightarrow K_2 \quad \kappa_3 \vdash M_3 \approx A_3 : K_1 \rightarrow K'_2}{< \kappa_1, \kappa_2, \kappa_3 > \vdash (M_1 \rightarrow M_2, M_3) \approx A_1 \text{ then either } A_2 \text{ or } A_3 : K_1 \rightarrow K_2 \cup K'_2}$$

$$\frac{\kappa_1 \vdash M_1 \approx A_1 : K_1 \rightarrow K'_1}{\rho \vdash \lambda x. M_1 \approx \text{abstraction of } K_1 A_1 : D \rightarrow F} \quad K_1 \leq F, K'_1 > FI$$

$$\frac{\rho \vdash M_1 \approx A_1 : D \rightarrow F}{\rho \vdash \text{let } X = M_1 \approx \text{furthermore}((\text{recursively}_X A_1 \text{ then bind } X : D \rightarrow D)}$$

$$\frac{\kappa \vdash M_1 \approx A_1 : K \rightarrow K}{\kappa \vdash \text{fix}(\lambda \alpha. M_1) \approx \text{unfolding } A_1 : K \rightarrow K}$$

$$\frac{\kappa_1 \vdash M_1 \approx A_1 : K_1 \rightarrow F \quad \kappa_2 \vdash M_2 \approx A_2 : K_2 \rightarrow K'_2}{\kappa_1, \kappa_2 \vdash M_1 M_2 \approx (A_1 \text{ and } A_2) \text{ then call} : K_1 \cup K_2 \rightarrow F} \quad K_1 = K'_2 \leq I$$

$$\frac{\kappa_1 \vdash M_1 \approx A_1 : K_1 \rightarrow F \quad \kappa_2 \vdash M_2 \approx A_2 : K_2 \rightarrow F}{\kappa_1, \kappa_2 \vdash M_1 M_2 \approx (A_1 \text{ and } A_2) \text{ then apply} : K_1 \cup K_2 \rightarrow F} \quad K_1 = K'_1 = K'_2 = F$$

$$\rho \vdash \rho I \approx \text{find } I : D \rightarrow F$$

**Fig. 14.** Rewriting rules for some of the  $\lambda$ -terms in theorem 1

- $A_1$  has type  $K_1 \rightarrow F$  And  $A_2$  has type  $K_2 \rightarrow K'_2$  with  $K_1 = K'_2 \leq I$ . In this case  $M$  rewrites to

$$\begin{aligned} & (A_1 \text{ and } A_2) \text{ then call} = \\ & \lambda\kappa\lambda\phi\lambda\pi\lambda\sigma.\phi(\pi, \sigma)(\lambda\kappa_1\lambda\kappa_2.(A_1(\kappa_1), A_2(\kappa_2))(\kappa))\triangleright \\ & \quad \lambda\kappa\lambda\phi\lambda\pi\lambda\sigma.\phi(\pi, \sigma)(\langle A_1(\kappa \downarrow 1), A_2(\kappa \downarrow 2) \rangle)\triangleright \\ & \quad \lambda\kappa.A_1(\kappa \downarrow 1)(A_2(\kappa \downarrow 2)) = \lambda\kappa_1\lambda\kappa_2.A_1(\kappa_1)(A_2(\kappa_2)) \end{aligned}$$

- $A_1$  has type  $K_1 \rightarrow F$  And  $A_2$  has type  $K_2 \rightarrow K'_2$  with  $K_1 = K'_2 \leq F$ .  $M$  rewrites to

$$\begin{aligned} & (A_1 \text{ and } A_2) \text{ then apply} = \\ & \lambda\kappa\lambda\phi\lambda\pi.\phi(\pi)(\lambda\kappa_1\lambda\kappa_2.(A_1(\kappa_1), A_2(\kappa_2))(\kappa))\triangleright \\ & \quad \lambda\kappa\lambda\phi\lambda\pi.\phi(\pi)(\langle A_1(\kappa \downarrow 1), A_2(\kappa \downarrow 2) \rangle)\triangleright \\ & \quad \lambda\kappa.A_1(\kappa \downarrow 1)(A_2(\kappa \downarrow 2)) = \lambda\kappa_1\lambda\kappa_2.A_1(\kappa_1)(A_2(\kappa_2)) \end{aligned}$$

- Finally, If  $M \equiv M_1M_2 \equiv \rho(I)$  then  $M$  rewrites to “find  $I$  (an axiom, last line in figure 14). Both have the same semantics, indeed.

**Definition 19.** Let  $S$  and  $S'$  be two denotational semantics description of a language  $\mathcal{L}$ . A 2-homomorphism  $t : \text{Act}_S \rightarrow \text{Act}'_{S'}$ , written as  $\text{Act}_S \gg \text{Act}'_{S'}$ , is a  $\Sigma$  – homomorphism from  $\text{Act}_S$  to  $\text{Act}'_{S'}$  where every carrier of  $\text{Act}_S$  is isomorphic to a subset of its corresponding carrier of  $\text{Act}'_{S'}$  and its replacement with that subset preserves the semantics of  $\text{Act}_S$ .

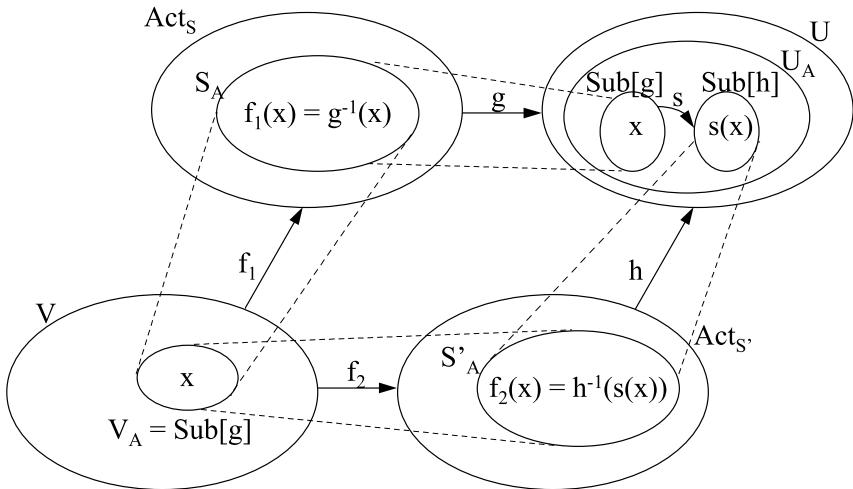
**Proposition 1.** A set  $C$  of the action algebras of all denotational semantics descriptions of a language  $\mathcal{L}$ , together with all 2-homomorphisms between them define a category of action algebras for  $\mathcal{L}$ .

**Lemma 1.** Let  $S$  be a denotational semantics description of a language  $\mathcal{L}$ , and let  $S'$  be another denotational semantics description of a language  $\mathcal{L}$  obtained from  $S$  by adding one semantic domain; then there exists a 2-homomorphism from  $\text{Act}_S$  to  $\text{Act}_{S'}$ .

**Proof Sketch.** When we add a semantic domain to a semantic description  $S$ , it may be classified as model dependent or language inherent. In the first case, the new domain occurs as a sort of  $\text{Act}_{S'}$ . As it does not occur at  $\text{Act}_S$ , the isomorphic condition still holds. In the case of the new domain be considered as language inherent, it may be a domain compatible with any of the denotational domains, and so it is not new indeed, otherwise it must be used in the definition of some, but not all, semantic functions. If it happens, the semantic function, where it occurs, may be considered as a refinement of its original definition in  $S$  (a kind of internalization of the algebraic construction present in  $S$ ) , thus, isomorphic to a sub-algebra of the corresponding carrier in  $S'$ .

**Lemma 2.** Let  $S$  and  $S'$  be two denotational semantics description of a language  $\mathcal{L}$ . Then there is an Action Algebra  $U$  such that  $\text{Act}_S \gg U$  and  $\text{Act}_{S'} \gg U$ .

**Proof.** Using Lemma 1 we build, step by step, Action Algebras from  $\text{Act}_S$  until the inclusion of all the sorts of  $\text{Act}_S$  not represented in  $\text{Act}_S$ .

**Fig. 15.** Pull-back diagram

**Lemma 3.** Let  $S$  and  $S'$  be two denotational semantics description of a language  $\mathcal{L}$ . Suppose that there is  $U$  such that  $g : \text{Act}_S \gg U$  and  $h : \text{Act}'_{S'} \gg U$ , then there is the Pull-Back  $V$  of  $g$  and  $h$  ( $V = Pb(g, h, U)$ ).

**Proof:** Let  $\Sigma$  be the common signature of  $\text{Act}_S$  and  $\text{Act}_{S'}$ .  $V$  is a  $\Sigma$ -Algebra, for we need only taking into account the sorts present in both  $\text{Act}_S$  and  $\text{Act}_{S'}$ . The carriers of  $V$  will be taken from the sub-algebras (in  $U$ ) determined by either  $g$  or  $h$  according the relationship between them. For the sake of clarity we will proceed with the construction for a sort  $A$  represented in both  $\text{Act}_S$  and  $\text{Act}_{S'}$  as illustrated in figure 15. As  $g$  and  $h$  are 2-homomorphisms we can say that  $Im[g](S_A)$  is isomorphic to a sub-algebra  $\text{Sub}[g]$  of  $U_A$ , the carrier of  $U$  with sort  $A$ , the same with respect to  $Im[h](S'_A)$  and  $\text{Sub}[h]$ . We argue that there is a homomorphism from  $\text{Sub}[g]$  into  $\text{Sub}[h]$  (or vice-versa), otherwise  $S$  and  $S'$  do not specify the same programming language. Let  $s$  be this homomorphism (from  $\text{Sub}[g]$  into  $\text{Sub}[h]$ ). Then the definition of the homomorphisms  $f_1 : V \rightarrow \text{Act}_S$  and  $f_2 : V \rightarrow \text{Act}_{S'}$  is as follows, when regarding the sort  $A$ . First we define  $V_A = \text{Sub}[g]$ , then for any  $x$  in  $V_A$ ,  $f_1(x) = g^{-1}(x)$  and  $f_2(x) = h^{-1}(s(x))$ . It can be seem that this construction, when done for the other sorts, yields two 2-homomorphisms ( $f_1$  and  $f_2$ ) from  $V$  to  $\text{Act}_S$  and  $\text{Act}_{S'}$  respectively such that  $f_1 \circ g = f_2 \circ h$ . The universality of  $V$  follows from the definition of  $f_1$  and  $f_2$  above.

**Theorem 2.** Given a family  $F$  of specifications for a language  $\mathcal{L}$  (over the same abstract syntax) there is an Action Algebra  $A(L, F)$ , such that, there is a 2-homomorphism from it into the Action Algebras associated to each specifications present in  $F$ . These 2-homomorphism are universal concerning each Action Algebra associated to each member of the family  $F$ .

**Proof:** Use lemma 2 and lemma 3 iteratively in order to build  $A(L, F)$ . It is important to note that the universality of each 2-homomorphism  $\alpha$ , from  $A(L, F)$  into each Action Algebra  $A$  associated to each member of  $F$ , comes from the universality of the Pull-Back construction in 3, here extended to limits of finite diagrams. Thus, the trivial Action Algebra cannot be considered as  $A(L, F)$ , unless it is already in  $F$ . For, if the trivial Action Algebra has a 2-homomorphism  $\beta$  from it into any Action Algebra member of  $F$  then it factors uniquely through  $\alpha$ , and this is possible only when  $\alpha = \beta$ .

**Definition 20.**  $A(\mathcal{L})$  is the Action Algebra  $A(L, F)$  when  $F$  is the family of all specifications of  $\mathcal{L}$ . We call  $A(\mathcal{L})$  the language algebra of  $\mathcal{L}$ ,  $\text{Lang}_{\mathcal{L}}$ . It formalizes any definition of what is called as "underlying conceptual analysys" or "Language inherent concepts".

In order to illustrate the result of Theorem 2 we present in figure 16 a continuation passing style denotational semantics definition of the sample language previously shown in figure 5. We have just added continuations to the

<p>Syntactic Domains:</p> <ul style="list-style-type: none"> <li><math>\Pi</math>: Prog(Programs)</li> <li><math>\Delta</math>: Decl(Declarations)</li> <li><math>\Sigma</math>: Stmt(Statements)</li> <li><math>E</math>: Expr(Expressions)</li> <li><math>I</math>: Ide (Identifiers)</li> <li><math>N</math>: Num(Numbers)</li> </ul> <p>Abstract Syntax:</p> <ul style="list-style-type: none"> <li><math>\Pi = \Delta ;' \Sigma ;'</math></li> <li><math>\Delta = I</math></li> <li><math>  I_1 ;' I_2 ;' = I</math></li> <li><math>  \Delta_1 ;' \Delta_2 ;'</math></li> <li><math>\Sigma = I ;' E</math></li> <li><math>  I_1 ;' I_2 ;'</math></li> <li><math>  \Sigma_1 ;' \Sigma_2 ;'</math></li> <li><math>  ;' \{ \Delta ;' \Sigma_1 ;' \} ;'</math></li> <li><math>E = E_1 +' E_2</math></li> <li><math>  N</math></li> <li><math>  I ;'</math></li> <li><math>I = \text{token} ;</math></li> <li><math>N = \text{token} ;</math></li> </ul>	<p>Semantic Domains:</p> <ul style="list-style-type: none"> <li><math>n</math>: Names ;</li> <li><math>i</math>: Integers ;</li> <li><math>l</math>: Locations ;</li> <li><math>s</math>: Stores = Locations <math>\rightarrow</math> Integers ;</li> <li><math>c</math>: CmdConts = Stores <math>\rightarrow</math> Stores ;</li> <li><math>p</math>: Procedures = Locations <math>\rightarrow</math> CmdConts <math>\rightarrow</math> CmdConts ;</li> <li><math>v</math>: Values = Locations + Procedures ;</li> <li><math>e</math>: Environments = Names <math>\rightarrow</math> Values;</li> </ul> <p>Semantic Functions:</p> <ul style="list-style-type: none"> <li><math>P</math>: Prog <math>\rightarrow</math> Stores <math>\rightarrow</math> Stores ;</li> <li><math>D</math>: Decl <math>\rightarrow</math> Environments <math>\times</math> Stores <math>\rightarrow</math> Environments <math>\times</math> Stores ;</li> <li><math>S</math>: Stmt <math>\rightarrow</math> Environments <math>\rightarrow</math> CmdConts <math>\rightarrow</math> CmdConts ;</li> <li><math>E</math>: Expr <math>\rightarrow</math> Environments <math>\rightarrow</math> Stores <math>\rightarrow</math> Integers ;</li> <li><math>I</math>: Ide <math>\rightarrow</math> Name ;</li> <li><math>N</math>: Num <math>\rightarrow</math> Integers ;</li> </ul>
<p>Semantic Equations:</p> <ul style="list-style-type: none"> <li><math>P[[ \Delta ;' \Sigma ;' ]] = S[[ \Sigma ]](D[[ \Delta ]](e,s), c), e = \lambda n. \perp, c = \lambda s. s;</math></li> <li><math>D[[ I ]](e, s) = &lt; e[I[[ I ]]] \leftarrow I, s   l \leftarrow \perp   &gt;, l = \text{free}(s) ;</math></li> <li><math>D[[ \Delta_1 ;' \Delta_2 ]](e,s) = (D[[ \Delta_1 ]]) o D[[ \Delta_2 ]](e,s) ;</math></li> <li><math>D[[ I_1 ;' I_2 ;' = \Sigma ]](e,s) = &lt; e[I[[ I_1 ]]] \leftarrow p, s &gt;, p(l) = S[[ \Sigma ]](e[I[[ I_2 ]]] \leftarrow l) ;</math></li> <li><math>S[[ I ;' E ]](e,c) = \lambda s. c(s   e[I[[ I ]]] \leftarrow E[[ E ]](e,s)) ;</math></li> <li><math>S[[ I_1 ;' I_2 ;' ]](e,c) = p(c), l = e(I[[ I_2 ]]), p = e(I[[ I_1 ]]) ;</math></li> <li><math>S[[ \Sigma_1 ;' \Sigma_2 ;' ]](e,c) = (S[[ \Sigma_1 ]](e) o S[[ \Sigma_2 ]](e))(c) ;</math></li> <li><math>S[[ \{ \Delta ;' \Sigma_1 ;' \} ]](e,c) = \lambda s. S[[ \Sigma_1 ]](e_1, c)(s_1), &lt; e_1, s_1 &gt; = D[[ \Delta ]](e,s) ;</math></li> <li><math>E[[ E_1 +' E_2 ]](e,s) = E[[ E_1 ]](e,s) + E[[ E_2 ]](e,s) ;</math></li> <li><math>E[[ N ]](e,s) = N[[ N ]] ;</math></li> <li><math>E[[ I ]](e,s) = s(e(I[[ I ]])) ;</math></li> <li><math>I[[ I ]] = \text{token to Names};</math></li> <li><math>N[[ N ]] = \text{token to Integers};</math></li> </ul>	

**Fig. 16.** Continuation-style Denotational Semantics

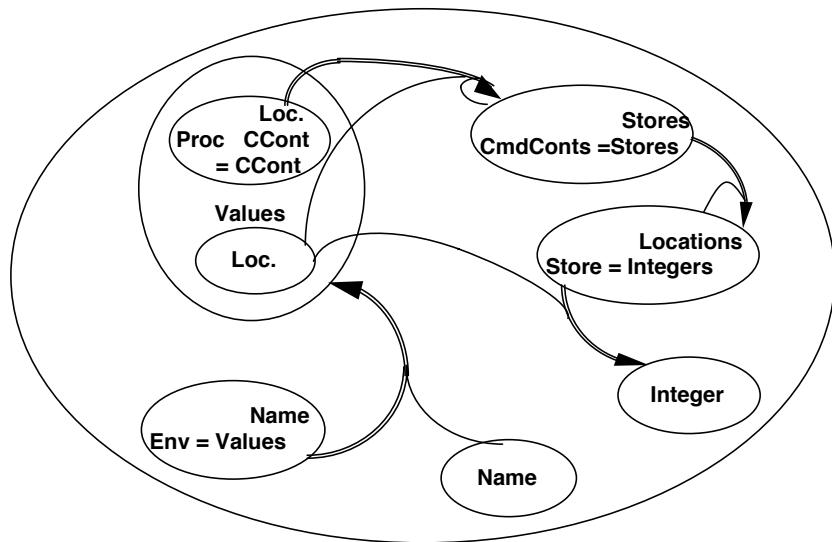


Fig. 17. *CmdCont* Domain

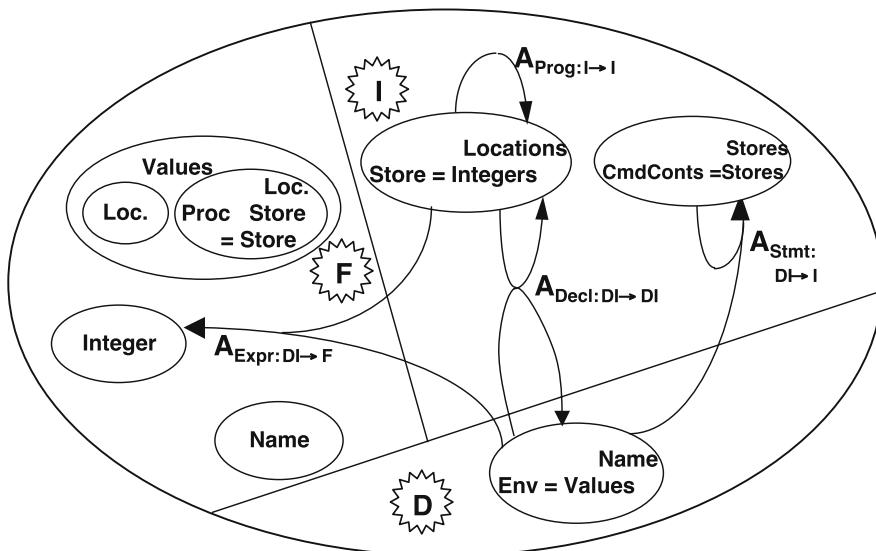


Fig. 18. Derived Action Algebra with Continuations

statements definition in order to keep it simple enough to be followed. According to Definition 16, the continuation domain,  $CmdCont$ , is model dependent and thus occurs as a sort of the Model Algebra defined for continuation semantics of  $\mathcal{L}$ , illustrated in Figure 17. From the Model Algebra, the Action Algebra shown in figure 18 comes easily and with it the matching with the Target Algebra of Action Semantics.

From both specifications of the sample language  $\mathcal{L}$ , we can see a 2-homomorphism from the direct style Action Algebra,  $ActS$ , into the continuation passing style Action Algebra,  $Act_{S'}$ . The carrier mapping comes from the identity function but the morphisms deserves some attention to the mapping of the Statement semantics,  $AStmt$ .

We can relate the continuation passing style semantics of a statement in  $\mathcal{L}$  to its direct style semantics by thinking that a Statement  $C$  followed by a continuation  $c$ , acting on a store  $s$ ,  $S'[[C]](c)(s)$  behaves exactly like the continuation  $c$  acting on the storage  $s$  affected by Statement  $C$ ,  $c(S[[C]](s))$ , resulting:

$$c(S[[C]](s)) = S'[[C]](c)(s)$$

In order to get the semantics of an isolated statement, we can assume that it is followed by a continuation that does nothing, like  $\lambda s.s$ . Thus, if we replace  $c$  for  $\lambda s.s$ , we have

$$\lambda s.s(S[[C]](s)) = S'[[C]](\lambda s.s)(s)$$

that gives

$$S[[C]](s) = S'[[C]](\lambda s.s)(s)$$

and abstracting on  $s$

$$S[[C]] = S'[[C]](\lambda s.s)$$

As  $AStmt$  comes from the statement semantic function  $S : Stmt \rightarrow Environments \rightarrow Stores \rightarrow Stores$ . Actually, we can say that  $S : Stmt \rightarrow AStmt$ , and analogously,  $S' : Stmt \rightarrow AStmt'$ . Hence, substituting  $AStmt$  and  $AStmt'$  for  $S$  and  $S'$ , respectively, we have:

$$AStmt = AStmt'(\lambda s.s)$$

From this example, we notice that the given direct style semantics of  $\mathcal{L}$  is a candidate for being its Language Algebra.

## 6 Conclusion

We have distinguished and precisely defined the concepts of model dependence and language inheritance. We have also showed that our separation between model dependent domains and language inherent domains is in agreement with the separation performed by Action Semantics. This distinction allows compiler generating systems based on Denotational Semantics to enhance their performance with the facet based architecture of Action Semantics and thus producing compilers that do not intertwine on their code the usage of model dependent and language

inherent handling structures. This result reinforces the idea of generating fast compilers from denotational semantics.

The separation between model dependent domains and language inherent domains and its agreement with action semantics is the most important contribution of this paper. The translation from denotational semantics to action semantics is useful to corroborate with the separation, but it never takes place indeed.

The elements of language inherent domains have, always, their behavior explicitly defined by the language designer by means of a  $\lambda$ -term. The model dependent domains generally have their behavior defined implicitly.

The model dependent domains may have their efficiency improved by enhancements made by the compiler generator, without affecting the language inherent ones. Their implicit behavior should be implemented by generic classes that would be gradually and independently optimized in a object oriented compiler generation framework, like in [3]. This, together with the fact that the object oriented paradigm already gives a special treatment to model dependent domains would suggest that the it is a natural and elegant way to produce fast compilers from denotational semantics. This seems to answer, at least partially, why Object Orientation, combined with good denotational semantics descriptions lead to good compilers.

## Acknowledgements

The authors would like to thank the editor and the two anonymous referees for the suggestions and improvements provided.

## References

1. Doh, K.-G., Schmidt, D.A.: Action Semantics-Directed Prototyping. *Computer Language* 19(4), 213–233 (1993)
2. Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification 2. EATCS Monographs in Theoretical Computer Science. Springer, Heidelberg (1990)
3. Guedes, L.C.C.: An Objected Oriented Model for Semantics Directed Compiler Generation. D.Sc. Thesis PUC-RIO (1995) (in Portuguese)
4. Goguen, J.A., Thatcher, J.W., Wagner, E.G.: An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types. In: Current Trends in Programming Methodology, vol. 4, Prentice-Hall, Englewood Cliffs (1979)
5. Goguen, J., Malcolm, G.: Algebraic Semantics of Imperative Programs. MIT Press, Cambridge (1996)
6. Lee, P.: Realistic Compiler Generation. MIT Press, Cambridge (1989)
7. Mosses, P.D.: Action Semantics. Cambridge tracts in Computer Science, vol. (26). Cambridge University Press, Cambridge (1992)
8. Mosses, P.D.: Abstract Semantic Algebras. In: Proc. of the IFIP Conference on Formal Description of Programming Concepts II, pp. 45–70. North-Holland Publishing Co., Amsterdam (1983)

9. Mosses, P.D.: Basic Abstract Semantic Algebras. In: Plotkin, G., MacQueen, D.B., Kahn, G. (eds.) *Semantics of Data Types 1984*. LNCS, vol. 173, pp. 87–107. Springer, Heidelberg (1984)
10. Mosses, P.D.: The Operational Semantics of Action Semantics. DAIMI PB-418, Computer Science Department Aarhus University (1992)
11. Palsberg, J.: Provably Correct Compiler Generation. Ph.D. Thesis Aarhus University (1992)
12. Pleban, U.F., Lee, P.: On the Use of LISP in Implementing Denotational Semantics. In: Proc. ACM Conf on LISP and Functional Programming, pp. 233–248 (1986)
13. Goguen, J.: A Hidden Agenda. *Theoretical Computer Science* 245, 55–101 (2000)
14. Guedes, L.C., Haeusler, E.H., Rangel, J.L.: Object Oriented Semantics Directed Compiler Generation: A Prototype. In: Mosses, P.D., Schwartzbach, M.I., Nielsen, M. (eds.) *TAPSOFT 1995*. LNCS, vol. 915, pp. 807–808. Springer, Heidelberg (1995)
15. Nielson, F., Nielson, H.R.: Two-level semantics and code generation. *Theoretical Computer Science* 56, 59–133 (1988)

# Mobile Processes and Termination<sup>\*</sup>

Romain Demangeon<sup>1</sup>, Daniel Hirschkoff<sup>1</sup>, and Davide Sangiorgi<sup>2</sup>

<sup>1</sup> ENS Lyon, Université de Lyon, CNRS, INRIA, France

<sup>2</sup> INRIA and Università di Bologna, Italy

**Abstract.** This paper surveys some recent works on the study of termination in a concurrent setting. Processes are  $\pi$ -calculus processes, on which type systems are imposed that ensure termination of the process computations. Two approaches are exposed. The first one draws on the method of logical relations, which has been extensively used in the analysis of sequential languages. The second approach exploits notions from term rewriting.

## 1 Introduction

It is a pleasure and a honour to write a paper in this volume dedicated to Peter D. Mosses. Peter has been a strong promoter of the semantics of programming languages (e.g., [Mos01, Mos04, Mos06]), in particular *structured semantics*. A carefully devised semantics makes the meaning of the language constructs clear and unambiguous, and it can be used to prove fundamental behavioural properties. In this paper we make use of structured operational semantics to prove properties of *termination* in the computations originated by systems of processes. We thus review a strand of work that we have pursued during the past few years whose goal is precisely termination in concurrency.

A term terminates if all its reduction sequences are of finite length. As far as programming languages are concerned, termination means that computation in programs will eventually stop. In computer science, termination has been extensively investigated in term rewriting systems [DM79, DH95] and  $\lambda$ -calculi [Gan80, Mit96], where strong normalization is a more commonly used synonym.

Termination is however interesting also in concurrency. For instance, if we interrogate a process, we may want to know that an answer is eventually produced (termination alone does not guarantee this since other security properties, e.g., deadlock-freedom [Kob98], are also involved, but termination would be the main ingredient in a proof). Similarly, when we load an applet we would like to know that the applet will not run for ever on our machine, possibly absorbing all the computing resources (a ‘denial of service’ attack). In general, if the lifetime of a process can be infinite, we may want to know that the process does not remain alive simply because of non-terminating internal

---

\* This work has been supported by the European Project “HATS” (contract number 231620), and by the french ANR projects “CHoCo” and “Complice”.

activity, and that, therefore, the process will eventually accept interactions with the environment.

We study termination in the setting of the  $\pi$ -calculus. This is a very expressive formalism. A number of programming language features can be encoded, including functions, objects, and state (in the sense of imperative languages) [SW01a]. As a consequence, however, the notoriously-hard problems of termination for these features hit the  $\pi$ -calculus too. Concurrency, then, adds a further dimension of complexity.

Two main forms of type techniques have been used to handle termination in the  $\pi$ -calculus. The first form makes use of logical relations, and appears in, e.g., [YBK01, San06]. Logical relations are well-known in functional languages and are used, in particular, to prove termination of typed  $\lambda$ -calculi. The second form of techniques appears in, e.g., [DS06b, DHKS07, DHS08], and borrows ideas from *term rewriting systems*. Roughly, termination is ensured by identifying a measure which decreases after finite steps of reduction.

The logical relation techniques allow one to treat processes that have a functional flavour. Intuitively, a channel is *functional* if it appears only once in input, and the input is replicated. In other words, the service offered by the channel is always available and does not change over time.

We show how to apply the logical relation technique to a small sublanguage of the  $\pi$ -calculus,  $\mathcal{P}^-$ . This is a non-deterministic language, with only asynchronous outputs, and in which all names are functional. One of the reasons for having to restrict the logical relation technique to  $\mathcal{P}^-$  is that we need several times the Replication Theorems (laws for the distributivity of replicated processes). These theorems hold only if the names are functional.

The language  $\mathcal{P}^-$  itself is not very expressive. It is however a powerful language for the termination property. We then show that the termination of  $\mathcal{P}^-$  implies that of a larger language,  $\mathcal{P}$ . For this, we use process calculus techniques, most notably techniques for behavioural preorders.

It remains an open problem how to directly apply logical relations to process languages broader than “functional” languages akin to  $\mathcal{P}^-$ .

On the other hand, with term-rewriting techniques we fail to capture non-trivial functional behaviours. For instance, it appears difficult to type the processes encoding the simply-typed  $\lambda$ -calculus — we discuss this briefly in Section 5.2. However the term-rewriting techniques allow us to handle sophisticated stateful processes.

The paper is thus made of two parts. The first, describing the logical relation techniques, is in Section 4. The proofs are only sketched. For more details, we refer to [San06], where, moreover, the language of processes is richer. For instance, here the only first-order value is unit, whereas in [San06] arbitrary first-order values are allowed.

The second part of the paper, dealing with the term-rewriting techniques, is treated in Section 5. This part is more brief: we only discuss the basic ideas, referring to the literature for the various extensions and enhancements that have been studied.

## 2 The $\pi$ -Calculus

To describe our contributions, we will work using the syntax given in Table 1, which is based on the process constructs of the standard (monadic)  $\pi$ -calculus [Mil99, SW01a].

**Table 1.** Syntax of processes

$a, b, c, d, \dots, x, y, z \dots$  *Names*

<i>Values</i>	
$v, w ::= a$	name
$\star$	unit value
<i>Processes</i>	
$M, N ::= \mathbf{0}$	nil process
$H[\tilde{v}]$	recursion call
$a(x). M$	input
$\overline{v}w. M$	output
$M \mid M$	parallel
$M + M$	sum
$\nu \tilde{a} M$	restriction
$H ::= X$	recursion variable
$\mu X(\tilde{x}). M$	recursive definition

---

Bound names, free names, and names of a process  $M$ , respectively written  $\text{bn}(M)$ ,  $\text{fn}(M)$ , and  $\text{n}(M)$ , are defined in the usual way. We do not distinguish  $\alpha$ -convertible terms. Unless otherwise stated, we also assume that, in any term, each bound name is different from the free names of the term and from the other bound names. In a statement, we sometimes say that a name is *fresh* to mean that it does not occur in the objects of the statement, like processes and actions. In particular, a name  $a$  is *fresh for*  $M$  if  $a$  does not occur in  $M$ . If  $R'$  is a subterm of  $R$  we say that  $R'$  is *guarded in*  $R$  if  $R'$  is underneath a prefix of  $R$ ; otherwise  $R'$  is *unguarded in*  $R$ . We use a tilde to indicate a tuple. All notations are extended to tuples in the usual way.

In a recursive definition  $\mu X(\tilde{x}). M$ , the recursion variable is  $X$  and the formal parameters are  $\tilde{x}$ . The actual parameters of a recursion are supplied in a recursion call  $H[\tilde{v}]$ . We require that, in a recursive definition  $\mu X(\tilde{x}). M$ , the only free recursion variable of  $M$  is  $X$ . This constraint simplifies some of our proofs, but can be lifted. Moreover, the recursion variable  $X$  should be guarded in the body  $M$  of the recursion.

When a recursion has no parameters we abbreviate  $\mu X(). R$  as  $\mu X.R$ , and calls  $(\mu X.R)[ ]$  and  $X[ ]$  as  $\mu X.R$  and  $X$ , respectively. For technical reasons, we find it convenient to manipulate a restriction operator that introduces several names at once.

A *first-order value* is a value that does not contain links. Examples are: an integer, a boolean value, a pair of booleans, a list of integers. To facilitate the reading of the proofs, in this paper the only first-order value is the **unit** value, written  $\star$ .

The monadic  $\pi$ -calculus serves as a basis for the type systems we shall present. We will sometimes adopt variants or restrictions of it, that we now briefly discuss.

The SOS rules for the transition relation of the processes of the calculus are presented in Table 2, where  $\alpha$  ranges over actions. (The symmetric versions of PAR-1, COM-1, CLOSE-1, and SUM-1 have been omitted.) They are the usual transition rules for the  $\pi$ -calculus, in the early style.

**Definition 1.** A process  $M$  diverges (or is divergent) if there is an infinite sequence of processes  $M_1, \dots, M_n, \dots$  with  $M_1 = M$ , such that, for all  $i$ ,

$$M_i \xrightarrow{\tau} M_{i+1}.$$

$M$  terminates (or is terminating), written  $M \in \text{TER}$ , if  $M$  is not divergent.

**Table 2.** Transition rules

INP: $a(x). M \xrightarrow{av} M\{v/x\}$	REC: $\frac{M\{\mu X(\tilde{x}). M/X\}\{\tilde{v}/\tilde{x}\} \xrightarrow{\alpha} M'}{(\mu X(\tilde{x}). M)[\tilde{v}] \xrightarrow{\alpha} M'}$
OUT: $\bar{a}v. M \xrightarrow{\bar{av}} M$	RES: $\frac{\nu \tilde{b} M \xrightarrow{\alpha} M'}{(\nu a, \tilde{b})M \xrightarrow{\alpha} \nu a M'} \quad a \notin \text{n}(\alpha)$
SUM-1: $\frac{M \xrightarrow{\alpha} M'}{M + N \xrightarrow{\alpha} M'}$	PAR-1: $\frac{M \xrightarrow{\alpha} M'}{M   N \xrightarrow{\alpha} M'   N'} \quad \text{if } \text{bn}(\alpha) \cap \text{fn}(N) = \emptyset$
COM-1: $\frac{M \xrightarrow{av} M' \quad N \xrightarrow{\bar{av}} N'}{M   N \xrightarrow{\tau} M'   N'}$	
OPEN: $\frac{\nu \tilde{b} M \xrightarrow{\bar{x}a} M'}{(\nu a, \tilde{b})M \xrightarrow{(\nu a)\bar{x}a} M'} \quad x \neq a$	
CLOSE-1: $\frac{M \xrightarrow{ab} M' \quad N \xrightarrow{\nu b \bar{a}b} N'}{M   N \xrightarrow{\tau} \nu b(M'   N')}$	if $b \notin \text{fn}(M)$

### The localised $\pi$ -calculus

For the study based on logical relations, we shall work in a *localised* calculus [Mer01], in which the recipient of a link cannot use it in input. Formally, in an input  $a(x).M$ , name  $x$  cannot appear free in  $M$  as subject of an input. (The subject of an input is the name at which the input is performed; for instance, the subject of  $a(x).M$  is  $a$ .) Locality has been found useful in practice – it is adopted by a number of experimental languages derived from the  $\pi$ -calculus, most notably Join [FG96] – and has also useful consequences on the theory [Mer01]. In this part of the paper, where we explore methods based on logical relations, locality is essential in the results involving logical relations: most of our results rely on it.

### Other process operators

Below in the paper, we shall also consider other  $\pi$ -calculus languages. They are defined from the operators introduced above, with transition rules as by Table 2, plus: *asynchronous output*,  $\overline{av}$ , that is, output without continuation; and *replication*,  $!M$ , which represents an infinite parallel composition of copies of  $M$  (replication will in particular be useful for the definition of the type systems discussed in Section 5). Their transition rules are:

$$\overline{av} \xrightarrow{\overline{av}} \mathbf{0} \quad \frac{M \mid !M \xrightarrow{\alpha} M'}{!M \xrightarrow{\alpha} M'}$$

In the paper, we will often move interchangeably between recursion and the *replication* construct, as they have the same expressiveness [SW01a].

## 3 The Simply-Typed Calculus

The grammar of types for the simply-typed  $\pi$ -calculus is:

$$T ::= \sharp T \mid \text{unit}$$

where the *connection type*  $\sharp T$  is the type of a link that carries tuples of values of type  $T$ , and *unit* is the only *first-order type* (that is, the type of a first-order value).

A *link* is a name of a connection type. A link is *first order* if it carries first-order values. It is *higher order* if it carries higher-order values (i.e., links). Note that ‘first-order name’ is different from ‘first-order link’: a first-order name has a type *unit*, whereas a first-order link has a type  $\sharp \text{unit}$ .

Our type system is à la Church, thus each name has a predefined type. We assume that for each type there is an infinite number of names with that type. We write  $x \in T$  to mean that the name  $x$  has type  $T$ . Similarly, each recursion variable  $X$  has a predefined tuple of types, written  $X \in \langle \tilde{T} \rangle$ , indicating the types of the arguments of the recursion. A judgment  $\vdash M$  says that  $M$  is a well-typed process; a judgment  $\vdash v : T$  says that  $v$  is a well-typed value of type  $T$ . For values  $v, w$  we write  $v : w$  to mean that  $v$  and  $w$  have the same type.

**Table 3.** Typing rules

$$\begin{array}{c}
\text{T-PAR : } \frac{\vdash M \quad \vdash N}{\vdash M \mid N} \qquad \qquad \text{T-SUM : } \frac{\vdash M \quad \vdash N}{\vdash M + N} \\
\\
\text{T-REC : } \frac{X \in \langle \tilde{T} \rangle \quad \tilde{x} \in \tilde{T} \quad \vdash \tilde{v} : \tilde{T} \quad \vdash M}{\vdash (\mu X(\tilde{x}). M)[\tilde{v}]} \\
\\
\text{T-RVAR : } \frac{X \in \langle \tilde{T} \rangle \quad \vdash \tilde{v} : \tilde{T}}{\vdash X[\tilde{v}]} \\
\\
\text{T-OUT : } \frac{\vdash v : \# T \quad \vdash w : T \quad \vdash M}{\vdash \overline{v}w. M} \qquad \text{T-INP : } \frac{\vdash v : \# T \quad x \in T \quad \vdash M}{\vdash v(x). M} \\
\\
\text{T-RES : } \frac{x_i \in \# T_i \text{ for some } T_i (1 \leq i \leq n)}{\vdash (\nu x_1 \dots x_n) M} \\
\\
\text{T-NIL : } \vdash \mathbf{0} \qquad \text{T-UNIT : } \vdash \star : \text{unit} \qquad \text{T-VAR : } \frac{x \in T}{\vdash x : T}
\end{array}$$


---

The typing rules are in Table 3. The typing rules for the operators of Section 2 (asynchronous output, replication) are similar to those of (standard) output and parallel composition.

A process is *closed* if all its free names have a connection type (i.e., they are links). The closed processes are the ‘real’ processes, those that, for instance, are supposed to be run, or to be tested. If a process is not closed, it has some names yet to be instantiated. A *closing substitution* for a process  $R$  is a substitution  $\sigma$  such that  $R\sigma$  is closed ( $R$  may already be closed, and  $\sigma$  may just rename some of its links).

The main point in simple types for the  $\pi$ -calculus is to guarantee that in all executions of a typable process, the type of values transmitted along a channel agrees with the type assigned to the channel. Simple types do not tell anything about termination of processes. The type systems we present in this paper will however be built on top of the type system for simple types.

## 4 Terminating Processes, via Logical Relations

In this section, we discuss how to isolate a subcalculus of the *localised*  $\pi$ -calculus in which all processes terminate by exploiting logical relations.

In sequential higher-order languages, it is well-known that termination may be broken by features such as self-applications, recursion, higher-order state. Our language of terminating processes is defined by four constraints. Three of them, mostly syntactic, are reported in Condition 2. The first condition is for recursive inputs; the second and third conditions control state. The last constraint – condition 1 of Definition 3 – controls self-applications.

We explain the new terminology used in the conditions. A name  $a$  appears free in *output position* in  $N$  if  $N$  has a free occurrence of  $a$  in an output prefix. An input is *replicated* if the input is inside the body of a recursive definition. Thus, a process  $M$  has free replicated first-order inputs if  $M$  contains a free first-order input inside the body of a recursive definition. For instance, if  $a$  is a first-order link then

$$\mu X. a(y). (\mathbf{0} \mid X)$$

has free replicated first-order inputs, whereas

$$\nu a (\mu X. a(y). (\mathbf{0} \mid X)) \mid b(z). \mathbf{0}$$

has not.

### Condition 2 (Termination constraints on the grammar)

1. Let  $\tilde{a} = a_1, \dots, a_n$ . In a process  $\nu \tilde{a} M$ , if  $a_i(x). N$  is a free input of  $M$ , then the following holds:
  - (a)  $a_i \in \tilde{a}$ ,
  - (b) names  $a_j$  with  $j \geq i$  do not appear free in output position in  $N$ .
2. In an higher-order input  $a(x). M$ , the continuation  $M$  does not contain free higher-order inputs, and does not contain free replicated first-order inputs.
3. In a recursive definition  $\mu X(\tilde{x}). M$ :
  - (a)  $\tilde{x}$  are first order,
  - (b)  $M$  has no unguarded output and no unguarded if-then-else.

Condition (1b) poses no constraints on occurrences of names not in  $\tilde{a}$ . The condition can be made simpler, but weaker, by requiring that names  $\tilde{a}$  do not appear free in output position in  $M$ .

To illustrate the meaning of condition (3a), consider a 1-place buffer, that receives values at a link  $a$  and retransmits them at a link  $b$ :

$$\mu X. a(x). \bar{b}x. X$$

This process respects the condition regardless of whether the values transmitted are first order or higher order. By contrast, a delayed 1-place buffer

$$\mu X(x). a(y). \bar{b}x. X[y],$$

which emits at  $b$  the second last value received by  $a$ , respects the condition only if the values transmitted are first order.

We say that a process  $M$  respects the constraints of Condition 2 to mean that  $M$  itself and all its process subterms respect the constraints.

**Definition 3 (Language  $\mathcal{P}$ ).**  $\mathcal{P}$  is the set of processes such that  $M \in \mathcal{P}$  implies:

1.  $M$  is typable in the simply-typed  $\pi$ -calculus;
2.  $\nu \tilde{a} M$  respects the constraints of Condition 2, where  $\tilde{a} = \text{fn}(M)$ .

**Theorem 1.** *All processes in  $\mathcal{P}$  terminate.*

The proof of this theorem is in two parts; the first one is sketched in Sections 4.1–4.3, the second one in Section 4.4. Throughout these sections, all processes and values are well typed, and substitutions map names onto values of the same type. Moreover, processes have no free recursion variables.

#### 4.1 $\mathcal{P}^-$ : Monadic Functional Non-deterministic Processes

We define a very constrained calculus  $\mathcal{P}^-$  whose processes will be proved to terminate using the technique of logical relations. We will then use  $\mathcal{P}^-$  to derive the termination of the processes of the language  $\mathcal{P}$  of Theorem 1 (Section 4.4).

The processes of  $\mathcal{P}^-$  are functional, that is, the input end of each link occurs only once, is replicated, and is immediately available (cf., the uniform-receptiveness discipline, [San99]). To emphasize the ‘functional’ nature of these processes, we use the (input-guarded) replication operator  $!a(x).M$  instead of recursion. Processes can however exhibit non-determinism, due to the presence of the sum operator. Outputs are asynchronous, that is, they have no continuations.

For the definition of  $\mathcal{P}^-$ , and elsewhere in the paper, it is useful to work up to structural congruence, a relation that allows us to abstract from certain details of the syntax of processes.

**Definition 4 (Structural congruence).** *Let  $R$  be a process of a language  $\mathcal{L}$  whose operators include parallel composition, restriction, replication, and  $\mathbf{0}$ . We write  $R \equiv_1 R'$  if  $R'$  is obtained from  $R$  by rewriting, in one step, a subterm of  $R$  using one of the rules below (from left to right, or from right to left)*

$$\begin{aligned} !R &= R \mid !R \\ \nu\tilde{a} \nu\tilde{b} R &= (\nu\tilde{a}, \tilde{b})R \\ (\nu\tilde{a}, a, b, \tilde{b})R &= (\nu\tilde{a}, b, a, \tilde{b})R \\ R_1 \mid R_2 &= R_2 \mid R_1 \\ R_1 \mid (R_2 \mid R_3) &= (R_1 \mid R_2) \mid R_3 \\ R \mid \mathbf{0} &= R \end{aligned}$$

(Note that if  $R \equiv_1 R'$  then  $R'$  need not be in  $\mathcal{L}$ .) Structural congruence,  $\equiv$ , is the reflexive and transitive closure of  $\equiv_1$ .

The definition of  $\mathcal{P}^-$  uses the syntactic categories of *processes*, *pre-processes*, and *resources*. The *normal forms* for processes, pre-processes and resources of  $\mathcal{P}^-$  are given in Table 4, where  $\text{in}(P)$  are the names that appear free in  $P$  in input position. Each new name is introduced with a construct of the form  $\nu a (!a(x).N \mid P)$  where the resource  $!a(x).N$  is the only process that can ever input at  $a$ . In the definition of resources, the constraint  $a \notin \text{fn}(M^{\text{NF}})$  prevents mutual recursion (calls of the replication from within its body).

Normal forms are not closed under reduction. For instance, if  $M^{\text{NF}} \xrightarrow{\tau} N$  then  $N$  may not be a process of the grammar in the table. However,  $N$  is structurally congruent to a normal form. We therefore define processes, resources,

**Table 4.** The normal forms

<i>Pre-processes</i>	
$P^{\text{NF}} ::= \nu a (I_a^{\text{NF}} \mid P^{\text{NF}})$	with $\text{fn}(I_a^{\text{NF}}) \cap \text{in}(P^{\text{NF}}) = \emptyset$
	$M^{\text{NF}}$
	$I_a^{\text{NF}}$
<i>Resources</i>	
$I_a^{\text{NF}} ::= !a(x). M^{\text{NF}}$	with $a \notin \text{fn}(M^{\text{NF}})$
<i>Processes</i>	
$M^{\text{NF}} ::= \nu a (I_a^{\text{NF}} \mid M^{\text{NF}})$	
	$M^{\text{NF}} + M^{\text{NF}}$
	$M^{\text{NF}} \mid M^{\text{NF}}$
	$\overline{v}w$
	$\mathbf{0}$
<i>Values</i>	
$v ::= a$	name
	*
	unit value

---

pre-processes by closing the normal forms with structural congruence. We need the reduction-closure property (Lemma 4) in later proofs.

**Definition 5 (Language  $\mathcal{P}^-$ ).** *The sets  $\mathcal{PR}$  of processes,  $\mathcal{RES}$  of resources, and  $\mathcal{P}^-$  of pre-processes are obtained by closing under  $\equiv$  the corresponding (well-typed) normal forms in Table 4.*

Thus  $M \in \mathcal{PR}$  if there is  $M^{\text{NF}}$  with  $M \equiv M^{\text{NF}}$ . Pre-processes include resources and processes (which explains why pre-processes are indicated with the symbol  $\mathcal{P}^-$ ). Pre-processes are ranged over by  $P, Q$ ; resources by  $I_a$ , processes by  $M, N$ . If  $\tilde{a}$  is  $a_1, \dots, a_n$  then  $\nu\tilde{a} (I_{\tilde{a}} \mid P)$  abbreviates  $\nu a_1 (I_{a_1} \mid \dots \nu a_n (I_{a_n} \mid P) \dots)$ , and similarly for  $\nu\tilde{a} (I_{\tilde{a}}^{\text{NF}} \mid P)$ .

## 4.2 Logical Relations on Processes

We recall the main steps of the technique of logical relations in the  $\lambda$ -calculus:

1. assignment of types to terms;

2. definition of a typed logical predicate on terms, by induction on the structure of types; the base case uses the termination property of interest;
3. proof that the logical terms (i.e., those in the logical predicate) terminate;
4. proof, by structural induction, that all well-typed terms are logical.

For applying logical relations to the  $\pi$ -calculus we follow a similar structure. Some of the details however are rather different. For instance, in the  $\pi$ -calculus an important role is played by a closure property of the logical predicate with respect to bisimilarity, and by the (Sharpened) Replication Theorems. Further, in the  $\lambda$ -calculus typing rules assign types to terms; in the  $\pi$ -calculus, by contrast, types are assigned to names. To start off the technique (step 1), we therefore force an assignment of types to the pre-processes. We use  $A$  to range over the types for pre-processes:

$$A ::= \diamond \mid b \_ \sharp T$$

where  $T$  is an ordinary type, as by the grammar in Section 3. If  $R, R' \in \mathcal{P}^-$  then  $R'$  is a normal form of  $R$  if  $R'$  is a normal form and  $R \equiv R'$ .

**Definition 6 (Assignment of types to pre-processes).** A normal form of a (well-typed) pre-process  $P$  is either of the form

- $\nu \tilde{a} (I_{\tilde{a}} \mid M)$ , or
- $\nu \tilde{a} (I_{\tilde{a}} \mid I_b)$ , with  $b \notin \tilde{a}$ .

In the first case we write  $P : \diamond$ , in the latter case we write  $P : b \_ T$ , where  $T$  is the type of  $b$ .

We define the logical predicate  $\mathcal{L}^A$  by induction on  $A$ .

**Definition 7 (Logical relations)**

- $P \in \mathcal{L}^\diamond$  if  $P : \diamond$  and  $P \in \text{TER}$ .
- $P \in \mathcal{L}^{a \_ \sharp \text{unit}}$  if  $P : a \_ \sharp \text{unit}$ , and for all  $v : \text{unit}$ ,

$$\nu a (P \mid \bar{a}v) \in \mathcal{L}^\diamond.$$

- $P \in \mathcal{L}^{a \_ \sharp T}$ , where  $T$  is a connection type, if  $P : a \_ \sharp T$ , and, for all  $b$  fresh for  $P$  and for all  $I_b \in \mathcal{L}^{b \_ T}$ ,

$$\nu b (I_b \mid \nu a (P \mid \bar{a}b)) \in \mathcal{L}^\diamond. \quad (1)$$

We write  $P \in \mathcal{L}$  if  $P \in \mathcal{L}^A$ , for some  $A$ .

In Definition 7, the most important clause is the last one. The process in (1) is similar to those used for translating function application into the  $\pi$ -calculus [SW01a]. Therefore a possible reading of (1) is that  $P$  is a function and  $I_b$  its argument. In (1),  $P$  does not know  $b$  (because it is fresh), and  $I_b$  does not know  $a$  (because it is restricted). However,  $P$  and  $I_b$  may have common free names, in output position.

### 4.3 Termination of $\mathcal{P}^-$

Before presenting the termination proof for  $\mathcal{P}^-$ , we present some general results on the  $\pi$ -calculus. We formulate them on  $\text{A}\pi_+$ : this is the  $\pi$ -calculus of Table 1, well-typed, without recursion, with only asynchronous outputs, and with the addition of replication. Here is the grammar of  $\text{A}\pi_+$ :

$$M ::= a(x). M \mid \bar{v}w \mid M + M \mid M \mid M \mid !M \mid \nu \tilde{a} M$$

where values  $v, w, \dots$  are as in Table 4.

**The Replication Theorems.** The proofs with the logical relations make extensive use of the Sharpened Replication Theorems [SW01a]. These express distributivity properties of private replications, and are valid for (strong) barbed congruence. We write  $M \downarrow_a$  if  $M \xrightarrow{\alpha} M'$  where  $\alpha$  is an input or an output along link  $a$ .

#### Definition 8 (Barbed congruence)

A relation  $\mathcal{R}$  on closed processes is a barbed bisimulation if whenever  $(M, N) \in \mathcal{R}$ ,

1.  $M \downarrow_a$  implies  $N \downarrow_a$ , for all links  $a$ ;
2.  $M \xrightarrow{\tau} M'$  implies  $N \xrightarrow{\tau} N'$  for some  $N'$  with  $(M', N') \in \mathcal{R}$ ;
3. the variants of (1) and (2) with the roles of  $M$  and  $N$  swapped.

Two closed processes  $M$  and  $N$  are barbed bisimilar if  $(M, N) \in \mathcal{R}$  for some barbed bisimulation  $\mathcal{R}$ .

Two processes  $M$  and  $N$  are barbed congruent,  $M \sim N$ , if  $C[M]$  and  $C[N]$  are barbed bisimilar, for every context  $C$  such that  $C[M]$  and  $C[N]$  are closed.

**Lemma 1.** Relation  $\sim$  is a congruence on  $\text{A}\pi_+$ .

**Lemma 2 (Sharpened Replication Theorems, for  $\text{A}\pi_+$ ).** Suppose  $a$  does not appear free in input position in  $M, N, N_1, N_2, \pi. N$ . We have:

1.  $\nu a (!a(x). M \mid !N) \sim !\nu a (!a(x). M \mid N)$ ;
2.  $\nu a (!a(x). M \mid N_1 \mid N_2) \sim \nu a (!a(x). M \mid N_1) \mid \nu a (!a(x). M \mid N_2)$ ;
3.  $\nu a (!a(x). M \mid \pi. N) \sim \pi. \nu a (!a(x). M \mid N)$ , where  $\pi$  is any input or output prefix;
4.  $\nu a (!a(x). M \mid (N_1 + N_2)) \sim \nu a (!a(x). M \mid N_1) + \nu a (!a(x). M \mid N_2)$ .

A wire is a process of the form  $!a(x). \bar{b}x$ . The lemma in this section says that, under certain conditions, wires do not affect termination.

**Lemma 3 (in  $\text{A}\pi_+$ ).** Suppose  $c'$  is a name that does not occur free in  $R$  in input position, and  $R$  only uses input-guarded replication. Then  $\nu c' (R \mid !c'(x). \bar{c}x)$  diverges iff  $R\{c/c'\}$  diverges.

**Closure properties.** We also need a few closure properties; to begin with, closure of the class  $\mathcal{P}^-$  of processes under reduction; then a closure of logical relations under  $\sim$ , closure of terminating processes under deterministic reduction, and a few properties of the logical predicates.

**Lemma 4 (Closure under reduction for  $\mathcal{P}^-$ ).**  *$R \in \mathcal{P}^-$  and  $R \xrightarrow{\tau} R'$  imply  $R' \in \mathcal{P}^-$ .*

**Lemma 5 (Closure under  $\sim$  for the logical relations).** *Suppose  $P, Q \in \mathcal{P}^-$ , and  $P \sim Q$ . If  $P \in \mathcal{L}^A$ , then also  $Q \in \mathcal{L}^A$ .*

**Lemma 6.** *If  $P \in \mathcal{L}$  then  $P \in \text{TER}$ .*

*Proof.* If  $P \in \mathcal{L}^\diamond$  then the result follows by definition of  $\mathcal{L}^\diamond$ . Otherwise  $P \in \mathcal{L}^{a-T}$ , for some  $a, T$ , and  $P$  cannot reduce.  $\square$

We write  $R \xrightarrow{\tau_d} R'$  if  $R \xrightarrow{\tau} R'$  and this is the only possible transition for  $R$  (i.e., for all  $\alpha, R''$  such that  $R \xrightarrow{\alpha} R''$ , we have  $\alpha = \tau$  and  $R' \equiv R''$ ).

**Lemma 7.** *If  $R \xrightarrow{\tau_d} R'$  and  $R' \in \text{TER}$  then also  $R \in \text{TER}$ .*

**Lemma 8.** *If  $a, b : T$  then  $!a(x). \bar{b}x \in \mathcal{L}^{a-T}$ .*

*Proof.* Suppose  $T = \sharp \text{unit}$ . Then

$$\nu a (!a(x). \bar{b}x \mid \bar{a}v) \xrightarrow{\tau_d} \sim \bar{b}v$$

(it terminates after one step), therefore by Lemma 7 and 5  $\nu a (!a(x). \bar{b}x \mid \bar{a}v) \in \text{TER}$ .

Otherwise, take a fresh  $c$  and any  $I_c$ , and consider the process

$$P \stackrel{\text{def}}{=} \nu c (I_c \mid \nu a (!a(x). \bar{b}x \mid \bar{a}c))$$

We have  $P \xrightarrow{\tau_d} \sim \nu c (I_c \mid \bar{b}c)$ , and the latter process cannot reduce further, therefore  $P \in \text{TER}$ , reasoning as above.  $\square$

**Lemma 9.** *Let  $c$  be a higher-order name. We have  $\nu b (I_b \mid \bar{b}c) \in \text{TER}$  iff  $(\nu b, c')(I_b \mid \bar{b}c' \mid !c'(x). \bar{c}x) \in \text{TER}$ , where  $c'$  is fresh.*

*Proof.* Follows from Lemma 3.  $\square$

**Relatively independent resources.** The final ingredient that we need for the main theorem (both its assertion and its proof) is that of relatively independent resources, roughly indicating a bunch of replicated processes without references to each other. We also need some lemmas that allow us to transform processes in the language so to have relatively independent resources.

**Definition 9.** *Resources  $I_{a_1}, \dots, I_{a_n}$  are said to be relatively independent if none of the names  $a_1, \dots, a_n$  appears free in output position in any of the resources  $I_{a_1}, \dots, I_{a_n}$ .*

A term  $P \in \mathcal{P}^-$  has relatively independent resources if, for all subterms  $P'$  of  $P$ , the resources that are unguarded in  $P'$  are relatively independent.

**Lemma 10.** *For each  $I_a \in \mathcal{RES}$  there is  $J_a \in \mathcal{RES}$  with  $I_a \sim J_a$  and  $J_a$  has relatively independent resources.*

*Proof.* Induction on the structure of a normal form of  $I_a$ , using the Replication Theorems.  $\square$

**Lemma 11.** *For each  $P \in \mathcal{P}^-$  there is  $Q \in \mathcal{P}^-$  with  $P \sim Q$  and  $Q$  has relatively independent resources.*

*Proof.* Similar to the previous lemma.  $\square$

**Lemma 12.** *For each  $P \in \mathcal{P}^-$  there is a normal form  $Q \in \mathcal{P}^-$  with  $P \sim Q$  and  $Q$  has relatively independent resources.*

*Proof.* If in the previous lemmas the initial process is in normal form, then also the transformed process is. The result then follows from the fact that  $\equiv \subseteq \sim$ .  $\square$

## Main theorem

**Theorem 2.** *Let  $\tilde{a} = a_1, \dots, a_n$ . Suppose that resources  $I_{a_1}, \dots, I_{a_n}$  are relatively independent, and  $I_{a_i} \in \mathcal{L}$  for each  $i$ . Then  $P : A$  and  $\text{in}(P) \cap \text{fn}(I_{\tilde{a}}) = \emptyset$  imply  $\nu \tilde{a} (I_{\tilde{a}} \mid P) \in \mathcal{L}^A$ .*

*Proof.* By Lemmas 12 and 5 we can assume that  $P$  is a normal form and has relatively independent resources. We proceed by induction on the structure of  $P$ . We call  $Q$  the process  $\nu \tilde{a} (I_{\tilde{a}} \mid P)$ . We only consider two cases.

–  $P = \bar{b}c$ .

In this case,  $A = \diamond$ . We have to show that  $Q \in \text{TER}$ . We have:

$$Q = \nu \tilde{a} (I_{\tilde{a}} \mid P) \sim \nu \tilde{a}' (I_{\tilde{a}'} \mid P) \stackrel{\text{def}}{=} Q'$$

where  $\tilde{a}' = \tilde{a} \cap \{b, c\}$  (here we exploit the fact that the resources are relatively independent).

There are 4 subcases:

- $\tilde{a}' = \emptyset$ . Then  $Q' \sim \bar{b}c$ , which is in TER.
- $\tilde{a}' = \{b\}$ . Then  $Q' \sim \nu b (I_b \mid \bar{b}c)$  and the latter process is in TER iff the process  $(\nu b, c')(I_b \mid !c'(x). \bar{c}x \mid \bar{b}c')$  is in TER (Lemma 9), where  $c'$  is fresh. And now we are done, exploiting the definition of  $\mathcal{L}$  on the type of  $b$ , for  $!c'(x). \bar{c}x \in \mathcal{L}$  (Lemma 8) and we know that  $I_b \in \mathcal{L}$ .
- $\tilde{a}' = \{c\}$ . Then  $Q' \sim \nu c (I_c \mid \bar{b}c)$  and the latter process is in TER because it cannot reduce.
- $\tilde{a}' = \{b, c\}$ . Then

$$\begin{aligned} Q' &\sim (\nu b, c)(I_b \mid I_c \mid \bar{b}c) \\ &\sim \nu c (I_c \mid \nu b (I_b \mid \bar{b}c)) \stackrel{\text{def}}{=} Q'' \end{aligned}$$

since  $c$  is fresh for  $I_b$  (the relatively-independence hypothesis). Now,  $Q''$  is in TER by definition of  $\mathcal{L}$  on higher types (precisely, the type of  $b$ ).

- $P = M_1 \mid M_2$ . (Thus  $P : \diamond$ )

We have to show that  $Q = \nu \tilde{a} (I_{\tilde{a}} \mid M_1 \mid M_2) \in \text{TER}$ . Using the Replication Theorems we have

$$Q \sim Q_1 \mid Q_2$$

where

$$Q_i \stackrel{\text{def}}{=} \nu \tilde{a} (I_{\tilde{a}} \mid M_i).$$

Now,  $Q \in \text{TER}$  iff each  $Q_i$  is so. The latter is true by the induction on the structure.  $\square$

**Corollary 1.** *If  $P \in \mathcal{P}^-$  then  $P \in \mathcal{L}$ .*

#### 4.4 Proofs Based on Simulation

The language  $\mathcal{P}^-$  is non-trivial, but not very expressive. It is however a powerful language for the termination property, in the sense that the termination of the processes in  $\mathcal{P}^-$  implies that of a much broader language, namely the language  $\mathcal{P}$  of Theorem 1.

We move to  $\mathcal{P}$  by progressively extending the language  $\mathcal{P}^-$ . The technique for proving termination of the extensions is as follows. The extensions define a sequence of languages  $\mathcal{P}_0, \dots, \mathcal{P}_{11}$ , with  $\mathcal{P}_0 = \mathcal{P}^-$ ,  $\mathcal{P}_{11} = \mathcal{P}$ , and  $\mathcal{P}_i \subset \mathcal{P}_{i+1}$  for all  $0 \leq i < 11$ . For each  $i$ , we exhibit a transformation  $\llbracket \cdot \rrbracket_i$ , defined on the normal forms of  $\mathcal{P}_{i+1}$ , with the property that a transformed process  $\llbracket M \rrbracket_i$  belongs to  $\mathcal{P}_i$  and  $\llbracket M \rrbracket_i \in \text{TER}$  implies  $M \in \text{TER}$ . We then infer the termination of the processes in  $\mathcal{P}_{i+1}$  from that of the processes in  $\mathcal{P}_i$ .

For this kind of proofs we use process calculus techniques, especially techniques for *simulation*. If  $R'$  simulates  $R$ , then  $R'$  can do everything  $R$  can, but the other way round may not be true. For instance,  $a. (b. c+d)$  simulates  $a. b$ , but the other way round is false. Simulation is not much interesting as a behavioural equivalence. Simulation is however interesting for reasoning about termination, and is handy to use because of its co-inductive definition.

**Definition 10.** *We say that a relation  $\mathcal{R}$  on closed processes is a strong simulation if  $(M, N) \in \mathcal{R}$  implies:*

- whenever  $M \xrightarrow{\alpha} M'$  there is  $N'$  such that  $N \xrightarrow{\alpha} N'$  and  $(M', N') \in \mathcal{R}$ .

*A process  $N$  simulates  $M$ , written  $M \preceq N$ , if for all closing substitutions  $\sigma$  there is a strong simulation  $\mathcal{R}$  such that  $(M\sigma, N\sigma) \in \mathcal{R}$ .*

Relation  $\preceq$  is reflexive and transitive, and is preserved by all operators of the  $\pi$ -calculus. Hence  $\preceq$  is a precongruence in all languages  $\mathcal{P}_i$ . An important property of  $\preceq$  for us is:

**Lemma 13.** *If  $M \preceq M'$  then  $M' \in \text{TER}$  implies  $M \in \text{TER}$ .*

Each language  $\mathcal{P}_i$  ( $i > 0$ ) is defined by exhibiting the additional productions for the normal forms of the processes and the resources of the language. We

only show a few examples of extensions and their correctness proofs. We refer to [San06] for the details.

$$M^{\text{NF}} ::= \dots \mid \bar{a}v. M^{\text{NF}}$$

*Proof.* Consider a transformation  $\llbracket \cdot \rrbracket$  that acts on outputs thus:

$$\llbracket \bar{b}v. M \rrbracket \stackrel{\text{def}}{=} \bar{b}v \mid \llbracket M \rrbracket .$$

and is an homomorphism elsewhere. Its correctness is given by the law

$$\bar{b}v. M \preceq \bar{b}v \mid M$$

and Lemma 13.  $\square$

$$I_a^{\text{NF}} ::= \dots \mid I_a^{\text{NF}} \mid I_a^{\text{NF}}$$

*Proof.* With the addition of this production, there can be several input-replicated processes at the same link. The correctness of this extension is inferred using the law

$$!a(x). R \mid !a(x). R' \preceq !a(x). (R \mid R') .$$

$\square$

$$I_{\tilde{a}}^{\text{NF}} ::= \dots \mid I_{\tilde{a}}^{\text{NF}} + I_{\tilde{a}}^{\text{NF}}$$

*Proof.* We use the law  $R_1 + R_2 \preceq R_1 \mid R_2$ .  $\square$

The addition of nested inputs and of recursion with first-order state are more difficult, though the basis of the proof is the same. Again, we refer to [San06]. Also, we refer to [San06] for more discussions on the importance of the clauses (2) and (3) of Condition 2 (on nesting of inputs and on state) for the termination of the processes of  $\mathcal{P}$ .

## 5 Methods Based on Rewriting Theory

In this section, we present a series of approaches to termination in the  $\pi$ -calculus based on rewriting techniques. In these works, a compositional type system is defined on top of processes, and the soundness of this analysis (that is, that every typable process terminates) is justified by exhibiting a well-founded order that decreases along reductions. The definition of this order typically exploits constructions like the lexicographic composition of two orders, or the multiset extension of an order.

We present type systems of increasing expressiveness, allowing one to type-check more and more processes. By essence, these type systems make it possible to reason about processes that exhibit non-trivial stateful structures, which is not the case for the analysis presented in the previous sections. On the other hand, it is not clear how to adapt these types to handle higher-order functions. So the two approaches (the one based on logical relations and the one based on term-rewriting) seem incomparable in terms of expressiveness.

We start by presenting the general approach in its simplest form. We then turn to refinements of the initial type system. We also review results about the complexity of type inference in these systems.

### 5.1 A Basic Type System

We begin with a very basic type system, that we hope conveys well the intuition about the term-rewriting approach. We call this system S1 (it corresponds to the first type system of [DS06a]).

The calculus we work with is the one of Table 1, but we find it convenient to have replicated inputs in place of recursion (see Section 2). Also, the constraint on locality (the obligation on the recipient of a name to use it in output only), which was necessary for logical relations, is not needed now, and can therefore be removed.

The type system extends the system of simple types for the  $\pi$ -calculus (cf. Table 3). Channels that are used according to type  $\sharp \text{unit}$  are written as CCS channels (we write  $a.M$  instead of  $a().M$  and  $\bar{a}.M$  instead of  $\bar{a}\star.M$ ). Typing judgements for values are of the form  $\vdash a : \sharp^k T$  where  $k$  is the *level* of  $a$  (we write in this case  $\text{lv}(a) = k$ ). Typing judgements for processes are of the form  $\vdash M : m$ , where  $M$  is a process and  $m$  is a natural number (the *weight* associated to  $M$ ).

The typing rules for system S1 are presented on Figure 5.1. (The presentation we give is not exactly the same as in [DS06a]. The present reformulation is equivalent, and we hope more clear.)

The type system controls divergences as follows. The weight associated to a process  $M$  is the maximum level of a channel which is used to emit a message in  $M$ , where outputs occurring under an operator of replication are ignored. For a replication to be typable, the level of the channel on which the replicated input occurs must be strictly greater than the level of all channels that are used in output in the continuation processes (again, outputs occurring under an inner replication are not taken into account), that is, this level must dominate the weight associated to the process (rule (**Rep1**)).

Consider the following examples of diverging processes:

$$M_1 \stackrel{\text{def}}{=} !a.\bar{a} \mid \bar{a} \quad M_2 \stackrel{\text{def}}{=} !a.\bar{b} \mid !b.\bar{a} \mid \bar{a} \quad M_3 \stackrel{\text{def}}{=} c(x).!a.\bar{x} \mid \bar{c}a \mid \bar{a}$$

They are ruled out by the typing rule for replication. If we write  $a \in \sharp^{k_a} \text{unit}$ ,  $b \in \sharp^{k_b} \text{unit}$  and  $c \in \sharp^{k_c} (\sharp^{k_x} \text{unit})$ , then type-checking the replication in  $M_1$  imposes  $k_a > k_a$ ; type-checking the replication in  $M_2$  imposes  $k_a > k_b > k_a$ ;

$$\begin{array}{c}
(\mathbf{Nil}) \frac{}{\vdash M : 0} \qquad (\mathbf{Res}) \frac{\vdash M : m \quad a \in T}{\vdash (\nu a) M : m} \\
\\
(\mathbf{Par}) \frac{\vdash M : m \quad \vdash N : n}{\vdash M \mid N : max(m, n)} \qquad (\mathbf{In}) \frac{\vdash a : \sharp^k T \quad x \in T \quad \vdash M : m}{\vdash \Gamma : a(x). Mm} \\
\\
(\mathbf{Out}) \frac{\vdash M : m \quad \vdash a : \sharp^k T \quad \vdash w : T}{\vdash \bar{a}w. M : max(m, k)} \\
\\
(\mathbf{Rep1}) \frac{\vdash M : m \quad \vdash a : \sharp^k T \quad x \in T \quad k > m}{\vdash !a(x). M : 0}
\end{array}$$

**Fig. 1.** Weight-based type system for termination

finally, type-checking the replication in  $M_3$  imposes  $k_x > k_x$ , as the typing rules for input and output have the effect of unifying  $k_a$  and  $k_x$ .

As announced, a typable process does not exhibit a diverging computation. This is expressed by the following result, that in turn relies on a subject reduction property of the type system.

### Theorem 3 (Soundness)

*If  $\vdash M : m$  for some  $m$ , then  $M$  terminates.*

*Proof (Sketch).* The main idea is to define a measure on processes that decreases along reductions. A communication involving a non replicated input prefix makes the process shrink. The interesting case is when a replicated input is triggered:

$$\bar{a}v. M \mid !a(x). N \xrightarrow{\tau} M \mid N\{v/x\} \mid !a(x). N .$$

In this case, by typability (rule **(Rep1)**), all outputs released by the consumption of the  $a(x)$  prefix (the outputs in  $N\{v/x\}$ ) are at a level strictly smaller than the level of  $a$ .

We hence define, as a measure on processes, the multiset of levels of names used in output (outputs occurring under a replication are ignored). This measure decreases strictly (for the multiset extension of the order on natural numbers) along every reduction of a typable process. Soundness is then proved by contradiction: an infinite sequence of reductions emanating from a typable process would induce an infinitely strictly decreasing sequence of multisets of natural numbers, which is impossible.

## 5.2 Enhancements of the Basic Type System

**Recursion.** A process like  $M_4 \stackrel{\text{def}}{=} !a. b. \bar{a}$  cannot be typed using the rules of Figure 5.1, because applying the rule for replication imposes  $k_a > k_a$ , if  $k_a$  is the level associated to channel  $a$ . In this process, the output on  $a$  is seen as a ‘recursive call’: triggering the replicated process located at  $a$  might lead to an

emission on  $a$  itself. However, this particular recursive call is innocuous, as an output on  $a$  and an output on  $b$  have to be consumed in order to produce a single output on  $a$ .

In order to be able to recognise some processes that exploit recursion as terminating, a further type system, called S2 here, is presented in [DS06a]. The basic idea is to keep the analysis we have presented above, but to treat *sequences of input prefixes* as a whole. For this, we manipulate typing judgements for processes of the form  $\vdash M : \mathcal{M}$ , where  $\mathcal{M}$  is a multiset of natural numbers (we use  $>_{mul}$  to denote the multiset extension of the standard ordering on natural numbers). Accordingly, the typing rules for parallel composition and for replicated inputs in system S2 are as follows ( $\uplus$  stands for multiset sum):

$$\begin{array}{c} (\mathbf{Par2}) \frac{\vdash M : \mathcal{M}_1 \quad \vdash N : \mathcal{M}_2}{\vdash M \mid N : \mathcal{M}_1 \uplus \mathcal{M}_2} \\ \\ (\mathbf{Rep2}) \frac{\vdash M : \mathcal{M} \quad \vdash a_1 : \sharp^{k_1} T_1 \quad \dots \quad \vdash a_q : \sharp^{k_q} T_q \quad \{k_1, \dots, k_q\} >_{mul} \mathcal{M}}{x_1 \in T_1 \quad \dots \quad x_q \in T_q \quad \vdash !a_1(x_1) \dots a_q(x_q). M : \emptyset} \end{array}$$

Notice how in rule **(Rep2)**, the sequence of input prefixes  $a_1(x_1) \dots a_q(x_q)$  is treated as a whole. In particular, process  $M_4$  can be type-checked: if we write  $\text{lv1}(b) = k_b$ , rule **(Rep2)** requires  $\{k_a, k_b\} >_{mul} \{k_a\}$ .

The soundness proof for system S2 is an adaptation of the one for the simpler type system seen above. There is essentially one additional technical difficulty: along the execution of a process, we need to reason about partially consumed sequences of input prefixes.

**Recursive Data Structures.** The limits of system S2 in terms of expressiveness appear when trying to type-check complex processes that mimic the behaviour of list-like or tree-like data structures.

More precisely, a process like

$$M_5 \stackrel{\text{def}}{=} !p(x, y). x. (\overline{y} \mid \overline{p}(x, y))$$

can typically arise in the encoding of the traversal of a list structure (for instance, the list can be used to implement a symbol table, in which data are stored and searched using concurrent accesses). We adopt here a *polyadic* version of the  $\pi$ -calculus, where tuples of names can be transmitted along channels: this represents a mild extension, and the type systems described above can be easily adapted to handle polyadicity.

In  $M_5$ , channel  $p$  can be seen as a node constructor,  $x$  as a node of the list and  $y$  as its successor. To represent a node  $a$  in the list having node  $b$  as successor, we trigger the replication by inserting process  $\overline{p}(a, b)$ . This has the effect of spawning an instance of the continuation, of the form  $a. (\overline{b} \mid \overline{p}(a, b))$ . This process intuitively trades a request on  $a$  for a request on  $b$  and reconstructs the node with an output on  $p$ , which corresponds to (a simplified representation of) the way we model in the  $\pi$ -calculus an access at  $a$  which is propagated to  $b$ .

The encoding of lists in the  $\pi$ -calculus moreover imposes that names  $x$  and  $y$  in  $M_5$  should have the same type, and hence the same level. This is intuitively the case because they represent the address of two nodes that play the same rôle in the (encoding of the) recursive structure. As a consequence, in  $M_5$ , the weight consumed is equal to the weight which is released when triggering the two input prefixes (on  $p$  and  $x$  – here, if  $p \in \sharp^{k_p}(\sharp^k \text{unit}, \sharp^k \text{unit})$ , the constraint in rule **(Rep2)** gives  $\{k_p, k\} >_{mul} \{k_p, k\}$ ). Process  $M_5$  hence cannot be typed using the type system presented above.

This motivates the definition of a more refined type system in [DS06a], that exploits a well-founded partial order between names. The main modification is as follows: for a replicated process to be typable, either we have  $\{k_1, \dots, k_q\} > \mathcal{M}$  (as in rule **(Rep2)**), or  $\{k_1, \dots, k_q\} = \mathcal{M}$  and the partial order between names decreases: we are trading inputs for outputs of the same level, but going down in the partial order.

This is the case in process  $M_5$ , provided we impose that  $x$  dominates  $y$  according to the partial order. In this more refined type system, partial order information is attached to the type of channels: for  $M_5$ , the type of  $p$  is of the form  $\sharp^{k_p}_{(1,2)}(\sharp^k \text{unit}, \sharp^k \text{unit})$ , which enforces that the first argument of an output on  $p$  to be greater than its second argument. This way,  $M_5$  can be accepted as terminating.

As we said, the typing hypothesis associated to  $p$  in the typing derivation for  $M_5$  induces some constraints on the usages of  $p$ : for an output of the form  $\overline{p}\langle a, b \rangle$  to be typable, partial order information should specify that  $a$  dominates  $b$ . Accordingly, in

$$M_6 \stackrel{\text{def}}{=} M_5 \mid \overline{p}\langle a, b \rangle \mid \overline{p}\langle b, c \rangle \mid \overline{p}\langle c, a \rangle ,$$

in order to type the outputs on  $p$ , we are bound to introduce a well-founded partial order on the (free) names  $a$ ,  $b$  and  $c$ . Since this relation is necessarily cyclic, we cannot type-check  $M_6$  (note that  $M_6$  is not diverging, but  $M_6 \mid \overline{a}$  is).

*From lists to trees.* The type system we have described above is further enriched in [DHS08]. In that more refined system, one is able to type-check processes corresponding to tree structures. An example is given by

$$M_7 \stackrel{\text{def}}{=} !p(x, y, z). x. (\overline{y} \mid \overline{z} \mid \overline{p}\langle x, y, z \rangle) .$$

Here,  $p$  is used to construct nodes of binary trees:  $x$  has two children,  $y$  and  $z$ . This extension leads to some technicalities, that are related to the fact that we must allow the weight to grow along the firing of a replication (a request on a node is propagated to several child structures).

**Typing Termination in the  $\lambda$ -calculus.** There exist encodings of the  $\lambda$ -calculus into the  $\pi$ -calculus, given a reduction strategy in the  $\lambda$ -calculus (cf. [SW01b]). The type system of Section 3 makes it possible to translate a typable (according to the simply typed  $\lambda$ -calculus) function of the  $\lambda$ -calculus into a

typable  $\pi$ -calculus process. Thus this provides a method to show that a  $\lambda$ -term does not exhibit divergences in the call by name or to call by value disciplines.

On the other hand, the same approach cannot be followed (at least not directly) using the term-rewriting-based type systems presented in this section. The argument is non-trivial. We discuss here that for the system S1.

**Lemma 14.** *There exists a simply typed  $\lambda$ -term whose call-by-value encoding into the  $\pi$ -calculus cannot be typed according to system S1.*

*Proof.* The call-by-value encoding of a  $\lambda$ -term  $M$  is given by a process  $\llbracket M \rrbracket_p$ , which is parametrised upon a *location channel*  $p$ . It is defined by the following clauses:

$$\begin{aligned} \llbracket \lambda x. M \rrbracket_p &\stackrel{\text{def}}{=} (\nu y) \overline{p} y. !y(x, q). \llbracket M \rrbracket_q & \llbracket x \rrbracket_p &\stackrel{\text{def}}{=} \overline{p} x \\ \llbracket M N \rrbracket_p &\stackrel{\text{def}}{=} (\nu q, r) (\llbracket M \rrbracket_q \mid \llbracket N \rrbracket_r \mid q(y). r(z). \overline{y}(z, p)) \end{aligned}$$

The call-by-name encoding is defined using similar ideas (see [SW01b]). In order for the encoding of a  $\lambda$ -term to be typable using S1, we must analyse the replicated inputs that are introduced when encoding  $\lambda$ -abstractions. It turns out that different usages of the same  $\lambda$ -calculus variable may induce constraints that make it impossible to apply rule **(Rep1)** of Figure 5.1.

For instance, if we consider the following typing judgement in the  $\lambda$ -calculus

$$f : (\sigma \rightarrow \tau) \rightarrow \tau \rightarrow \tau, u : \sigma \rightarrow \tau, v : \sigma \vdash (f \ \lambda x. (f u (u v))) : \tau \rightarrow \tau ,$$

we observe that the encoding of  $(f \ \lambda x. (f u (u v))) : \tau \rightarrow \tau$  cannot be typed according to S1. The interested reader can write down the details of the encoding. It appears, in doing that, that the abstraction on  $x$  is translated into a replicated input on some channel  $y$ , and that the type of  $y$  must be unified with the type of some  $y_1$  channel which is used to encode the application  $(u v)$ . Since  $y_1$  is used in output within the scope of the replication on  $y$ , we get a form of ‘recursive call’, which prevents us from typing the process. It can be noted that the extensions of S1 presented above do not help either in typing the encoding of this  $\lambda$ -term.

### 5.3 On the Complexity of Type Inference

[DHKS07] presents an analysis of the problem of type inference for (some of) the systems we have discussed above.

**Theorem 4.** *The type inference problem for system S1 is polynomial.*

*Proof (Sketch).* Type inference in S1 boils down to searching for cycles in a directed graph: we construct a graph by associating a vertex to every name used in the process, and we draw an edge from  $a$  to  $b$  to represent the constraint  $1\text{vl}(a) > 1\text{vl}(b)$ . This algorithm is implemented in [Kob07], and in [Bou08], where a more expressive variant of S1, described in [DHS08], is also implemented.

On the contrary, we prove in [DHKS07] the following result on the type inference problem for system S2 of Section 5.2:

**Theorem 5.** *The type inference problem for system S2 is NP-complete.*

The intuitive reason for this result is that in rule **(Rep2)**, we are using a multiset ordering, which leads to a combinatorial number of possible level assignments. This allows us to reduce the problem **3SAT** to the problem of type inference.

We describe the main ideas behind the reduction, because it provides a good illustration of how system S2 works.

*Proof.* An instance  $\mathcal{I}$  of **3SAT** is made of  $n$  clauses  $(C_i)_{i \leq n}$  of three literals each,  $C_i = l_i^1, l_i^2, l_i^3$ . Literals are possibly negated propositional variables taken from a set  $V = \{v_1, \dots, v_m\}$ . The problem is to find a mapping from  $V$  to  $\{\text{True}, \text{False}\}$  such that, in each clause, at least one literal is set to True.

Given an instance  $\mathcal{I}$  of **3SAT**, we describe how we build an instance of the problem of the type inference. We fix a particular name `true`. To each variable  $v_k \in V$ , we associate two names  $x_k$  and  $x'_k$ , and define the process

$$M_k \stackrel{\text{def}}{=} !\text{true}.\text{true}.\overline{x_k}.\overline{x'_k} \mid !x_k.x'_k.\overline{\text{true}} .$$

We then consider a clause  $C_i = \{l_i^1, l_i^2, l_i^3\}$  from  $\mathcal{I}$ . For  $j \in \{1, 2, 3\}$  we let  $y_i^j = x_k$  if  $l_i^j$  is  $v_k$ , and  $y_i^j = x'_k$  if  $l_i^j$  is  $\neg v_k$ . We then define the process

$$N_i \stackrel{\text{def}}{=} !y_i^1.y_i^2.y_i^3.\overline{\text{true}} .$$

We call  $\mathcal{I}_t$  the problem of finding a typing derivation in S2 for the process  $M \stackrel{\text{def}}{=} M_1 \mid \dots \mid M_m \mid N_1 \mid \dots \mid N_n$ .

The constraints corresponding to the existence of such a solution are as follows (the level associated to name `true` is noted  $t$ ):

– for each  $M_k$ :

$$(\text{lvl}(x_k) = t \wedge \text{lvl}(x'_k) < t) \vee (\text{lvl}(x'_k) = t \wedge \text{lvl}(x_k) < t) . \quad (2)$$

– for each  $N_i$ :

$$t \leq \text{lvl}(y_i^1) \vee t \leq \text{lvl}(y_i^2) \vee t \leq \text{lvl}(y_i^3) . \quad (3)$$

We now prove that ' $\mathcal{I}_t$  has a solution' is equivalent to ' $\mathcal{I}$  has a solution'.

First, if  $\mathcal{I}$  has a solution  $S : V \rightarrow \{\text{True}, \text{False}\}$  then fix  $t = 2$ , and set  $\text{lvl}(x_k) = 2, \text{lvl}(x'_k) = 1$  if  $v_k$  is set to True, and  $\text{lvl}(x_k) = 1, \text{lvl}(x'_k) = 2$  otherwise. We check easily that condition (2) is satisfied; condition (3) also holds because  $S$  is a solution of  $\mathcal{I}$ .

Conversely, if  $\mathcal{I}_t$  has a solution, then we deduce a boolean mapping for the literals in the original **3SAT** problem. Since constraint (2) is satisfied, we can set  $v_k$  to True if  $\text{lvl}(x_k) = t$ , and False otherwise. We thus have that  $v_k$  is set to True iff  $\text{lvl}(x_k) = t$ , iff  $\text{lvl}(x'_k) < t$ . Hence, because constraint (3) is satisfied, we have that in each clause  $C_i$ , at least one of the literals is set to True, which shows that we have a solution to  $\mathcal{I}$ .

In [DHKS07], we also introduce a variant of system S2, which is strictly more expressive, and which relies on algebraic comparisons between multisets of names (instead of comparisons using only the multiset ordering). We show that type inference for this variant is polynomial. However, extending this variant with partial order information (as in Section 5.2), in order to allow processes where the weight does not decrease along the triggering of a replication, is more difficult than in the case of system S2.

## 6 Conclusions

We have discussed two type-based approaches for guaranteeing termination of  $\pi$ -calculus processes. The first one uses the method of logical relations, transplanted from functional languages. The second approach exploits notions from term-rewriting theory. We now give some additional comments about the results we have presented.

*Logical relations techniques.* In Theorem 1, we have established termination for  $\mathcal{P}$ , the simply-typed (localised)  $\pi$ -calculus subject to three syntactic conditions that constrain recursive inputs and state. In the proof, we have first applied the logical relation technique to a subset of processes with only functional names, and then we have extended the termination property to the whole language by means of techniques of behavioural preorders.

The termination of  $\mathcal{P}$  implies that of various forms of simply-typed  $\lambda$ -calculus: usual call-by-name, call-by-value, and call-by-need, but also enriched  $\lambda$ -calculi such as concurrent  $\lambda$ -calculi [DCdLP94],  $\lambda$ -calculus with resources and  $\lambda$ -calculus with multiplicities [BL00]. Indeed all encodings of  $\lambda$ -calculi into  $\pi$ -calculus we are aware of, restricted to simply-typed terms, are also encodings into  $\mathcal{P}$ . The  $\lambda$ -calculus with resources,  $\lambda^{\text{res}}$ , is a form of non-deterministic  $\lambda$ -calculus with explicit substitutions and with a parallel composition. Substitutions have a multiplicity, telling us how many copies of a given resource can be made. Due to non-determinism, parallelism, and the multiplicity in substitutions (which implies that a substitution cannot be distributed over a composite term), a direct proof of termination of  $\lambda^{\text{res}}$ , with the technique of logical relations, although probably possible, is non-trivial.

We have only considered the simply-typed  $\pi$ -calculus – the process analogous of the simply-typed  $\lambda$ -calculus. It should be possible to adapt our work to more complex types, such as those of the polymorphic  $\pi$ -calculus [Tur96, SW01a] – the process analogous of the polymorphic  $\lambda$ -calculus.

We have applied the logical relation technique only to a small set of ‘functional’ processes. Then we have had to use ad hoc techniques to prove the termination of a larger language. To obtain stronger results, and to extend the results more easily to other languages (for instance, process languages with communication of terms such as the Higher-Order  $\pi$ -calculus) a deeper understanding of the logical relation technique in concurrency would seem necessary.

*Term-rewriting techniques.* The various type systems we have presented in Section 5 all share the same general approach, that somehow is more syntactic

than the proof strategy based on logical relations. Indeed, the central idea is to design the termination analysis based on the structure of terms, in order to be able to define a measure that decreases along reductions of processes. As we have shown, this approach turns out to provide a great flexibility, making it possible to combine various tools from rewriting theory to type-check processes.

A further development of this approach has been studied in [RDS09], where the rewriting-based methods are applied to study termination in versions of the  $\pi$ -calculus that feature constructs for higher-order communication and, more generally, for the manipulation of process terms. This should open the way in particular for the adaptation of our results to recent proposals for models of distributed programming, where forms of code mobility are provided. This is left for future work.

Another interesting and natural direction for extensions of these studies is to work on the integration of the two approaches we have described. The processes that we can type in the first approach, although they may exhibit non-determinism, have a definite functional flavour. The second approach based on rewriting techniques has a more limited expressiveness on “functional” processes, but allows us to type more non-trivial “imperative” processes. We are currently working on combining the two approaches, with the goal of being able to handle systems in which both functional and imperative processes appear.

*Acknowledgements.* We would like to thank Yuxin Deng, who has collaborated with us on this line of research and made several important contributions. We have also benefited from comments and interactions with Naoki Kobayashi.

## References

- [BL00] Boudol, G., Laneve, C.:  $\lambda$ -calculus, multiplicities and the  $\pi$ -calculus. In: Plotkin, G., Stirling, C., Tofte, M. (eds.) *Proof, Language and Interaction: Essays in Honour of Robin Milner*, MIT Press, Cambridge (2000)
- [Bou08] Boutillier, P.: Implementation of a hybrid type system for termination in the  $\pi$ -calculus. Training period report, ENS Lyon (2008)
- [DCdLP94] Dezani-Ciancaglini, M., de Liguoro, U., Piperno, U.: Fully abstract semantics for concurrent  $\lambda$ -calculus. In: Hagiya, M., Mitchell, J.C. (eds.) *TACS 1994*. LNCS, vol. 789, pp. 16–35. Springer, Heidelberg (1994)
- [DH95] Dershowitz, N., Hoot, C.: Natural termination. *Theoretical Computer Science* 142(2), 179–207 (1995)
- [DHKS07] Demangeon, R., Hirschkoff, D., Kobayashi, N., Sangiorgi, D.: On the Complexity of Termination Inference for Processes. In: Barthe, G., Fournet, C. (eds.) *TGC 2007 and FODO 2008*. LNCS, vol. 4912, pp. 140–155. Springer, Heidelberg (2008)
- [DHS08] Demangeon, R., Hirschkoff, D., Sangiorgi, D.: Static and Dynamic Typing for the Termination of Mobile Processes. In: Proc. of IFIP TCS 2008. IFIP, vol. 273, pp. 413–427. Springer, Heidelberg (2008)
- [DM79] Dershowitz, N., Manna, Z.: Proving termination with multiset orderings. *Communications of the ACM* 22(8), 465–476 (1979)

- [DS06a] Deng, Y., Sangiorgi, D.: Ensuring Termination by Typability. *Information and Computation* 204(7), 1045–1082 (2006)
- [DS06b] Deng, Y., Sangiorgi, D.: Ensuring termination by typability. *Inf. Comput.* 204(7), 1045–1082 (2006)
- [FG96] Fournet, C., Gonthier, G.: The Reflexive Chemical Abstract Machine and the Join calculus. In: Proc. 23th POPL. ACM Press, New York (1996)
- [Gan80] Gandy, R.O.: Proofs of strong normalization. In: To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism. Academic Press, London (1980)
- [Kob98] Kobayashi, N.: A partially deadlock-free typed process calculus. *ACM Transactions on Programming Languages and Systems* 20(2), 436–482 (1998)
- [Kob07] Kobayashi, N.: TyPiCal: Type-based static analyzer for the Pi-Calculus (2007), <http://www.kb.ecei.tohoku.ac.jp/~koba/typical/>
- [Mer01] Merro, M.: Locality in the  $\pi$ -calculus and applications to object-oriented languages. PhD thesis, Ecoles des Mines de Paris (2001)
- [Mil99] Milner, R.: Communicating and Mobile Systems: the  $\pi$ -Calculus. Cambridge University Press, Cambridge (1999)
- [Mit96] Mitchell, J.C.: Foundations for Programming Languages. MIT Press, Cambridge (1996)
- [Mos01] Mosses, P.D.: The varieties of programming language semantics. In: Bjørner, D., Broy, M., Zamulin, A.V. (eds.) PSI 2001. LNCS, vol. 2244, pp. 165–190. Springer, Heidelberg (2001)
- [Mos04] Mosses, P.D.: Exploiting labels in structural operational semantics. *Fundam. Inform.* 60(1-4), 17–31 (2004)
- [Mos06] Mosses, P.D.: Formal semantics of programming languages: - an overview -. *Electr. Notes Theor. Comput. Sci.* 148(1), 41–73 (2006)
- [RDS09] Demangeon, R., Hirschkoff, D., Sangiorgi, D.: Termination in Higher-Order Concurrent Calculi. In: Proc. of FSEN 2009 (to appear, 2009)
- [San99] Sangiorgi, D.: The name discipline of uniform receptiveness. *Theo. Comp. Sci.* 221, 457–493 (1999)
- [San06] Sangiorgi, D.: Termination of processes. *Mathematical Structures in Computer Science* 16(1), 1–39 (2006)
- [SW01a] Sangiorgi, D., Walker, D.: The  $\pi$ -calculus: a Theory of Mobile Processes. Cambridge University Press, Cambridge (2001)
- [SW01b] Sangiorgi, D., Walker, D.: The  $\pi$ -calculus: a Theory of Mobile Processes. Cambridge University Press, Cambridge (2001)
- [Tur96] Turner, N.D.: The polymorphic pi-calculus: Theory and Implementation. PhD thesis, Department of Computer Science, University of Edinburgh (1996)
- [YBK01] Yoshida, N., Berger, M., Honda, K.: Strong normalisation in the  $\pi$ -Calculus. In: 16th Annual IEEE Symposium on Logic in Computer Science (LICS 2001), pp. 311–322. IEEE Computer Society Press, Los Alamitos (2001)

# An Action Semantics Based on Two Combinators

Kyung-Goo Doh<sup>\*</sup> and David A. Schmidt<sup>\*\*</sup>

Hanyang University, Ansan, South Korea  
Kansas State University, Manhattan, Kansas, USA  
`doh@hanyang.ac.kr, das@ksu.edu`

**Abstract.** We propose a naive version of action semantics that begins with a selection of “transient” and “persistent” facets, each characterized as a partial monoid. Yielders are defined as operations on the monoids’ values, and actions extract values from the facets, give them to yielders, and place the results into facet output. Actions are composed with a primary combinator, `andthen`, which can be specialized for multiple facet flows, and the choice combinator, `or`. Using big-step-style deduction rules, we give the semantics of yielders and actions, and we introduce a weakening rule and a strengthening rule, which let us compose actions with different facet domain-codomains. We also introduce *Mosses abstraction*, a lambda-abstraction variant that improves the readability of action-semantics definitions. Finally, we exploit the subsort (subtype) structure within Mosses’s unified algebras to use the deduction rules as both a typing definition as well as a semantics definition. Partial evaluation techniques are applied to type check and compile programs.

## 1 Introduction

Peter Mosses developed action semantics [9,10,11,12,13,15,16] as an antidote to the complexity of denotational semantics definitions, which use lambda-abstraction and application to model all possible language features and definitional styles. This leads to problems like the complete rewrite of a definition when moving from “direct style” to “continuation style” [14,20].

The key innovation within action semantics is the *facet* — an “active” semantic domain, a kind of value stream, that connects one action to another. A typical imperative language has a facet that represents the flow of temporaries, a facet that represents binding (environment/symbol-table) flow, and a facet that portrays the threading of primary store. Facets are analogous to Strachey’s characteristic domains for denotational semantics [21].

*Actions* operate on facets. An action is a “computational step,” a kind of state-transition function. Actions consume facet values and produce facet values. (In

---

\* This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology(MEST) / Korea Science and Engineering Foundation(KOSEF), R11-2008-007-01003-0.

\*\* Supported by NSF ITR-0326577.

the case of the store, an action updates the store and passes it to the next action). A language’s action set defines the language’s computational capabilities. When one action is composed with another, the facet flows must be directed to make one action affect another; combinators are used to direct the flows. *Yielders* are the “noun phrases” within actions, evaluating to the data values that are computed upon by the actions.

The end result is an expressive, high-level, data-flow-like semantics. It is simplistic to describe action semantics as a typed combinatory logic, but similarities do exist.

It is enlightening to review the evolution of action semantics from Mosses’s first papers on abstract semantic algebras [9,10,11] to unified algebras [12] to “linguistic-style” action semantics [13,15,16,22]. In Mosses’s early papers, one sees algebraic laws for infix combinators ( $; , ! []$ , etc.) that define flow of individual facets, as well as precisely stated combinations that direct multiple facet flows. The combinators evolved into a technical English of conjunctions (*and*, *then*, *hence*, *tentatively*, *furthermore*, *before*, etc.) that express a variety of flows of facet combinations [2,13,16].

Although the end result is a powerful, general-purpose, language-semantics toolkit, there is value in having a “lightweight” version of action semantics for tutorial purposes. In this paper, we propose a naive version of action semantics that begins with a selection of “transient” and “persistent” facets, each characterized as a partial monoid. Yielders are defined as operations on the monoids’ values, and actions extract values from the facets, give them to yielders, assemble results, and place the results into facet output. Actions are composed with a primary combinator, *andthen*, which can be specialized for multiple facet flows. There is the choice combinator, *or*.

Using big-step-style deduction rules, we give the semantics of yielders and actions, and we introduce a weakening rule and a strengthening rule, which let us compose actions with different facet domain-codomains. We also reintroduce Mosses-style lambda-abstraction, which we call *Mosses abstraction*, as a readability aid [9]. Finally, we exploit the subsort (subtype) structure within Mosses’s unified algebras [12] to use the deduction rules as both a typing definition as well as a semantics definition. Partial evaluation techniques [1] are applied to type check and compile programs.

## 2 Facets

A facet is a collection of data values, a Strachey-like characteristic domain [21], that are handled similarly, especially with respect to computational lifetime.

Formally defined, a *facet* is a (partial) monoid, that is, a set of values,  $S$ , a (partial) associative operation,  $\circ : S \times S \rightarrow S$ , and an identity element from  $S$ . Composition  $\circ$  defines how to combine or “glue” two values.

A facet is classified as *transient* or *persistent*, based on the lifetime (*extent*) of its values. A transient facet’s values are short-lived and are produced, copied, and consumed during a program’s computation; a persistent facet’s are

long-lived, fixed, data structures that are referenced and updated (and not produced, copied, and consumed).

Here are the facets we employ in this paper. First, we define by induction these sets of basic values:

$$\begin{aligned}\text{Identifier} &= \text{identifiers} \\ \text{Action} &= \text{text of actions} \\ \text{Cell} &= \text{storage locations} \\ \text{Int} &= \text{integers} \\ \text{Expressible} &= \text{Cell} \cup \text{Int}\end{aligned}$$

$$\begin{aligned}\mathcal{F} &= \text{List}(\text{Transient}) \\ \text{Transient} &= \mathcal{F} \cup \mathcal{D} \cup \text{Expressible} \cup \text{Closure} \\ \text{Closure} &= \text{Set}(\text{Transient}) \times \text{Identifier} \times \text{Action} \\ \mathcal{D} &= \text{Set}(\text{Identifier} \times \text{Denotable}) \\ \text{Denotable} &= \text{Transient} \\ \mathcal{I} &= \text{Cell} \rightarrow \text{Storable} \\ \text{Storable} &= \text{Int}\end{aligned}$$

The facets in this paper use the basic-value sets:

1. *functional facet*: The monoid of most-transient values is written  $\mathcal{F} = (\mathcal{F}, :, \langle \rangle)$ . The value set is sequences (lists) of transients, e.g.,  $\langle 2, \text{cell99}, \{(x, \text{cell99})\} \rangle$  is a three-transient sequence. Composition,  $:$ , is sequence append, and identity is the empty sequence,  $\langle \rangle$ . Functional-facet values have a brief extent, and the facet is a transient facet.
2. *declarative facet*: Sets of bindings are modelled by the monoid,  $\mathcal{D} = (\mathcal{D}, +, \{ \})$ ; The value set consists of finite sets of pairs,  $\rho = \{(I_0, n_0), (I_1, n_1), \dots, (I_m, n_m), \dots\}$ , where each such set defines a function (that is, each  $I_j$  is distinct). Composition is binding override: For values  $\rho_1$  and  $\rho_2$ , we define  $\rho_1 + \rho_2 = \rho_2 \cup \{(I_j = n_j) \in \rho_1 \mid I_j \notin \text{domain}(\rho_2)\}$  —  $\rho_2$ 's bindings take precedence over  $\rho_1$ 's. Identity is the empty set of bindings. Bindings are readily generated and copied — the facet is a transient facet.
3. *imperative facet*: Stores belong to the monoid,  $\mathcal{I} = (\mathcal{I}, *, [\ ])$ . The value set consists of finite functions,  $\sigma = [\ell_0 \mapsto n_0, \ell_1 \mapsto n_1, \dots, \ell_k \mapsto n_k]$ , each  $\ell_i \in \text{Cell}$  and  $n_i \in \text{Storable}$ . Composition,  $\sigma_1 * \sigma_2$ , is function union (provided that the functions' domains are disjoint) — a partial operation. The identity is the empty map. Since the imperative facet denotes persistent store, the facet is persistent.

A computation step (action) may require values from more than one facet, so we define a compound facet as a monoid of finite sets of facet elements, at most one element per facet: For *distinct* facets,  $\mathcal{F} = (F, \circ_F, id_F)$  and  $\mathcal{G} = (G, \circ_G, id_G)$ , define the compound facet as

$$\begin{aligned}\mathcal{FG} &= (\{\{f, g\} \mid f \in F, g \in G\}, \circ_{FG}, \{id_F, id_G\}) \\ &\quad \text{where } \{f_1, g_1\} \circ_{FG} \{f_2, g_2\} = \{f_1 \circ_F f_2, g_1 \circ_G g_2\}\end{aligned}$$

That is, a compound-facet value is a set of singleton-facet values, and composition is applied on the individual values in the respective sets based on facet affiliation. The construction allows compound facets like  $\mathcal{DI}$  but not  $\mathcal{FFDI}$ ; the latter case must be folded into  $\mathcal{FDI}$  by using the respective composition operators for  $\mathcal{F}$  and  $\mathcal{D}$ .

The “empty” compound facet is the *basic* (“control” [13]) facet, and it is the one-element monoid,  $\mathcal{B}$ . We use  $\Gamma, \Delta, \Sigma$  to stand for compound facet values.

We can embed a (compound) facet value into a larger compound facet by adding identity values: For element  $f = \{f_0, f_1, \dots, f_m\} \in \mathcal{F}_0 \mathcal{F}_1 \cdots \mathcal{F}_m$ , we embed  $f$  into  $\mathcal{F}_0 \mathcal{F}_1 \cdots \mathcal{F}_m \mathcal{G}_0 \mathcal{G}_1 \cdots \mathcal{G}_n$  as  $f \cup \{id_{G_0}, id_{G_1}, \dots, id_{G_n}\}$ .

Embedding is a technical device that lets us compose any two facet values together: For  $f$  and  $g$ , we define  $f \circ g$  by unioning their facet domains, embedding each of  $f$  and  $g$  into the unioned-facet domain, and composing the embedded elements in the unioned monoid.

We define a “strict union” of two (compound) facet values as this partial function: for  $\{f_1, \dots, f_m, g_1, \dots, g_n\} \in \mathcal{F}_1 \cdots \mathcal{F}_m \mathcal{G}_1 \cdots \mathcal{G}_n$  and  $\{f_1, \dots, f_m, h_1, \dots, h_p\} \in \mathcal{F}_1 \cdots \mathcal{F}_m \mathcal{H}_1 \cdots \mathcal{H}_p$ ,

$$\{f_1, \dots, f_m, g_1, \dots, g_n\} \cup \{f_1, \dots, f_m, h_1, \dots, h_p\} = \{f_1, \dots, f_m, g_1, \dots, g_n, h_1, \dots, h_p\}$$

That is, two facet values are unioned only if they agree on the values of their shared facets. In a similar manner, we define “facet restriction” and “facet subtraction”:

$$\{f_1, \dots, f_m, g_1, \dots, g_n\} \downarrow_{\mathcal{F}_1 \cdots \mathcal{F}_m \mathcal{H}_1 \cdots \mathcal{H}_p} = \{f_1, \dots, f_m\}$$

$$\{f_1, \dots, f_m, g_1, \dots, g_n\} \downarrow_{\sim \mathcal{F}_1 \cdots \mathcal{F}_m \mathcal{H}_1 \cdots \mathcal{H}_p} = \{g_1, \dots, g_n\}$$

### 3 Yielders

An action semantics requires operations on values carried within a facet and operations on the facets themselves. The former are called *yielders* and the latter are called *actions*.<sup>1</sup> Yielders are embedded within actions in a semantics definition; for this reason, they compute on transients.

Yielders are interesting because their arguments can often be computed at earlier binding times (compile-time, link-time) than run-time. Type checking, constant folding, and partial evaluation can be profitably applied to yielders, as we investigate in Section 11.

In our naive version of action semantics, we define yielders via big-step operational-semantics rules. Figure 1 presents a sample collection. Within a rule, read the configuration,  $\Gamma \vdash y : \Delta$ , as stating, “yielder  $y$  consumes inputs  $\Gamma$  to produce outputs  $\Delta$ .”

For example,  $(\text{add } (\text{find } x) \text{ it})$  is a yelder that adds the value bound to  $x$  in the declarative facet to the incoming transient in the functional facet. One possible derivation, for the functional, declarative facets,  $\langle n_1 \rangle, \{(x, n_0), (z, n_2)\}$ , goes as follows:

$$\frac{\{(x, n_0), (z, n_2)\} \vdash \text{find } x : n_0 \quad \langle n_1 \rangle \vdash \text{it} : n_1}{\{\langle n_1 \rangle, \{(x, n_0), (z, n_2)\}\} \vdash \text{add } (\text{find } x) \text{ it} : \text{add}(n_0, n_1)}$$

---

<sup>1</sup> In the “linguistic” version of action semantics [13,16,22], yielders are “noun phrases” and actions are “verb phrases.” But this distinction is not clearcut, e.g., “give (the denotable bound to  $x$ )” versus “find  $x$ ,” so we deemphasize this approach.

*Functional-facet yielders:*

primitive constant:  $\vdash k : k$

n-ary operation (e.g., addition):  $\frac{\Gamma \vdash y_1 : \tau_1 \quad \Delta \vdash y_2 : \tau_2}{\Gamma \cup \Delta \vdash \text{add } y_1 y_2 : \text{add}(\tau_1, \tau_2)}$

indexing:  $\frac{1 \leq i \leq m}{\langle \tau_1, \dots, \tau_m \rangle \vdash \#i : \tau_i}$  Note: it abbreviates  $\#1$

sort filtering:  $\frac{\Gamma \vdash y : \Delta \quad \Delta \leq T}{\Gamma \vdash \text{is}T y : \Delta}$  where  $\leq$  is defined in Section 10

*Declarative-facet yielders:* binding lookup, binding creation, and copy:

$$\frac{(l, \tau) \in \rho}{\rho \vdash \text{find } l : \tau} \quad \frac{\Gamma \vdash y : \tau}{\Gamma \vdash \text{bind } l y : \{(l, \tau)\}} \quad \frac{}{\rho \vdash \text{currentbindings} : \rho}$$

**Fig. 1.** Yielders

$\cup$  combines the input requirements of the component yielders in the consequent sequent.

Note there is a derivation for  $\text{is}T y$  exactly when  $y$  yields a value that belongs to sort (type)  $T$ .

## 4 Actions

Actions compute on facets in well-defined steps. In particular, actions enumerate the steps taken upon persistent-facet values like stores, databases, and i/o buffers.

There are “structural” actions that hand facet values to yielders and place the yielders’ results in facets; there are actions that operate on persistent-facet values; and there are actions that define and apply closures containing action-text. Figure 2 presents a sample action set, whose behaviors are defined with big-step deduction rules. Read  $\Gamma \vdash a \Rightarrow \Delta$  as asserting that action  $a$  receives facets  $\Gamma$  and produces facets  $\Delta$ .

Of the structural actions,  $\text{give}_G y$  hands yielder  $y$  its inputs and places its outputs into the facet stream named by  $G$ .  $\text{complete}$  is an empty computation step.

Actions **lookup**, **update**, and **allocate** define computation steps on the persistent facet,  $I$ ; a store value must be provided as input. The rule for **allocate** shows that the action produces a functional-facet value ( $c$ ) as well as an updated store.

The last two rules in the Figure portray closure construction and application. Assuming that yielder  $y$  evaluates as  $\Gamma \vdash y : \Delta$ , then  $\Gamma \vdash \text{recabstract}_G I y a : [\Delta \downarrow G, I, a]_G$  yields a closure that holds the  $G$ -facet portion of  $\Delta$ , the closure’s name,  $I$ , and the unevaluated action,  $a$ . Later,  $\text{exec } y_1 y_2$  evaluates  $y_1$  to the closure, evaluates  $y_2$  to an argument,  $\tau$ , and evaluates the closure body,  $a$ , to  $\langle \tau \rangle \circ (\Delta \cup (\Gamma \downarrow \sim G)) \circ \{(I, [\Delta, I, a]_G)\} \vdash a \Rightarrow \Sigma$ , that is, input  $\tau$  is composed with the data,  $\Delta$ , saved in the closure along with the facets within  $\Gamma$  that are allowed as the inputs to  $a$ . Name  $I$  rebinds to the same closure for recursive calls.

*Structural actions:*

$$\frac{\Gamma \vdash y : \Delta \quad \Delta \in \mathcal{G}}{\Gamma \vdash \text{give}_{\mathcal{G}} y \Rightarrow \Delta} \quad \text{where } \mathcal{G} \text{ names the facet that receives value } \Delta$$

$$\frac{}{\vdash \text{complete} \Rightarrow \text{completing}} \quad \text{where } \text{completing} \text{ is the sole element in the basic-facet monoid}$$

*Imperative-facet actions:*

$$\frac{\Gamma \vdash y : c \quad c \leq \text{Cell}}{\Gamma \cup \sigma \vdash \text{lookup } y \Rightarrow \langle \sigma(c) \rangle} \quad \frac{\Gamma_1 \vdash y_1 : c \quad c \leq \text{Cell} \quad \Gamma_2 \vdash y_2 : \tau \quad \tau \leq \text{Storable}}{\Gamma_1 \cup \Gamma_2 \cup \sigma \vdash \text{update } y_1 y_2 \Rightarrow \sigma[c \mapsto \tau]}$$

$$\frac{c \notin \text{domain}(\sigma)}{\sigma \vdash \text{allocate} \Rightarrow \langle c \rangle, \sigma * [c \mapsto ?]} \quad \text{Note: the output belongs to compound facet, } \mathcal{FI}$$

*Closure yielder and action:*

$$\frac{\Gamma \vdash y : \Delta}{\Gamma \vdash \text{recabstract}_{\mathcal{G}} I y a : [\Delta \downarrow_{\mathcal{G}}, I, a]_{\mathcal{G}}} \quad \text{where } \mathcal{G} \text{ names only transient facets}$$

$$\frac{\begin{array}{c} \Gamma_1 \vdash y_1 : [\Delta, I, a]_{\mathcal{G}} \\ \Gamma_2 \vdash y_2 : \tau \\ \Gamma = \Gamma_1 \cup \Gamma_2 \end{array}}{\Gamma \vdash \text{exec } y_1 y_2 \Rightarrow \Sigma} \quad \langle \tau \rangle \circ (\Delta \cup (\Gamma \downarrow_{\sim \mathcal{G}})) \circ \{(I, [\Delta, I, a]_{\mathcal{G}})\} \vdash a \Rightarrow \Sigma$$

Note:  $y_2$  is optional.

**Fig. 2.** Actions

$$\text{weaken-L: } \frac{\Gamma \vdash a \Rightarrow \Delta}{\Sigma \cup \Gamma \vdash a \Rightarrow \Delta}$$

$$\text{strengthen-R: } \frac{\Gamma \vdash a \Rightarrow \Delta \quad \Gamma \cup \sigma = \Gamma}{\Gamma \vdash a \Rightarrow \Delta \cup \sigma} \quad \text{where } \sigma \in \mathcal{I}$$

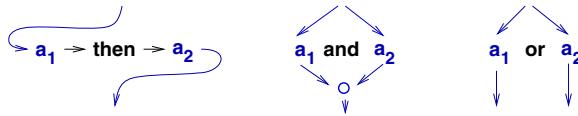
**Fig. 3.** Weakening and strengthening rules

For example, `abstractD f currentbindings (giveF(add (find x) it))` defines a statically scoped closure,  $f$ , that adds  $x$ 's value in the scope of definition to the argument supplied at the point of application.

Actions are polymorphic depending on the inputs and outputs of their embedded yielders, and we need rules to assemble compound actions. They are listed in Figure 3. The *weaken-L* rule states that an action can consume more facets than what are needed to conduct the action, and no harm occurs. The *strengthen-R* rule states that an action whose input includes a persistent value,  $\sigma$ , passes forwards that value unaltered (provided that the action did not itself alter the value — recall that  $\Delta \cup \sigma$  is “strict union,” defined iff  $\Delta$  holds no  $\mathcal{I}$ -value distinct from  $\sigma$ ).

Here is an example. We can derive that

$$\{(x, 2)\} \vdash \text{give}_F(\text{find } x) \Rightarrow 2$$

**Fig. 4.** Facet flows

Using the *weaken-L* rule, we deduce

$$\langle \rangle, \{(x, 2)\}, \sigma_0 \vdash \text{give}_F(\text{find } x) \Rightarrow 2$$

which shows that the empty sequence of transients and the persistent store do not alter the outcome. The *strengthen-R* rule lets us deduce that the store propagates unaltered:

$$\langle \rangle, \{(x, 2)\}, \sigma_0 \vdash \text{give}_F(\text{find } x) \Rightarrow 2, \sigma_0$$

## 5 Combinators

When two actions are composed, there are three possible patterns of a facet's flow: sequential, parallel, and conditional; see Figure 4. Parallel flow finishes by composing the outputs from the two component actions by monoid composition. Conditional flow allows only one action to produce the output.

The flows are modelled by the combinators, *then*, *and*, and *or*, respectively; here are their semantics:

$$\frac{\Gamma \vdash a_1 \Rightarrow \Delta \quad \Delta \vdash a_2 \Rightarrow \Sigma}{\Gamma \vdash a_1 \text{ then } a_2 \Rightarrow \Sigma} \quad \frac{\Gamma \vdash a_1 \Rightarrow \Delta_1 \quad \Gamma \vdash a_2 \Rightarrow \Delta_2}{\Gamma \vdash a_1 \text{ and } a_2 \Rightarrow \Delta_1 \circ \Delta_2} \quad \frac{\Gamma \vdash a_i \Rightarrow \Delta \quad i \in \{1, 2\}}{\Gamma \vdash a_1 \text{ or } a_2 \Rightarrow \Delta}$$

At this point, we have a kind of combinatory logic for the imperative facet,  $\mathcal{I}$ :  $a_1 \text{ then } a_2$  defines composition,  $a_2(a_1 \sigma)$ ;  $a_1 \text{ and } a_2$  defines an S-combinator,  $S a_1 a_2 \sigma = a_1(\sigma) \circ a_2(\sigma)$ ; the *L-weaken* rule defines a K-combinator,  $(ax)\sigma = ax$ ; and the *R-strengthen* rule defines an I-combinator,  $a \sigma = \sigma$ .

When actions consume multiple facets, it is likely that different flows are required for the individual facets. The standard example is command composition,  $C_1; C_2$ , where the incoming set of bindings (scope/symbol table) is given in parallel to both  $C_1$  and  $C_2$ , and the incoming store is threaded sequentially through  $C_1$ , which updates it and passes it to  $C_2$ . Such concepts are crucial to language semantics, and denotational semantics employs lambda-abstractions to encode such flows; in contrast, action semantics makes the flows primitive and explicit.

Our naive action semantics uses the combinator,  $\text{and}_{\mathcal{G}}\text{then}$ , where  $\mathcal{G}$  denotes the (compound) facet that is passed in parallel (by *and*) and all other facets are passed sequentially (by *then*). Here is its definition:

$$\frac{\Gamma \vdash a_1 \Rightarrow \Delta_1 \quad (\Gamma \downarrow_{\mathcal{G}}) \cup (\Delta_1 \downarrow_{\sim \mathcal{G}}) \vdash a_2 \Rightarrow \Delta_2}{\Gamma \vdash a_1 \text{ and}_{\mathcal{G}}\text{then } a_2 \Rightarrow \Delta_1 \downarrow_{\mathcal{G}} \circ \Delta_2}$$

In particular, *then* abbreviates  $\text{and}_{\emptyset}\text{then}$  and *and* abbreviates  $\text{and}_{\text{AllFacets}}\text{then}$ . When we omit the subscript and write  $\text{andthen}$ , we mean  $\text{and}_{\mathcal{D}}\text{then}$ , that is, only the declarative facet,  $\mathcal{D}$ , is consumed in parallel.

With *andthen* and *or*, we can readily model most mainstream language concepts.

Expression: $E ::= \text{k} \mid E_1 + E_2 \mid N$
Command: $C ::= N := E \mid C_1; C_2 \mid \text{while } E \text{ do } C \mid D \text{ in } C \mid \text{call } N(E)$
Declaration: $D ::= \text{val } I = E \mid \text{var } I = E \mid \text{proc } I_1(I_2) = C \mid \text{module } I = D \mid D_1; D_2$
Name: $N ::= I \mid N.I$
Identifier: $I$

Fig. 5. Example language syntax

## 6 Action Equations

A language’s action semantics is a set of equations, defined inductively on the language’s syntax. For the syntax in Figure 5, we define one valuation function for each syntax domain, one equation for each syntactic construction. Each valuation function has an arity that lists the facets that may be consumed and must be produced. For example, expressions are interpreted by the valuation function,  $\text{evaluate} : \text{Expression} \rightarrow \mathcal{DI} \rightarrow \mathcal{F}$ , which indicates that an expression might require the declarative and imperative facets to produce a functional-facet value. Figure 6 shows the action equations for the language described in Figure 5. We follow Mosses-Watt-style action notation, which elides the semantic brackets from single nonterminals, e.g.,  $\text{evaluate } E$  rather than  $\text{evaluate}[E]$ .

Wherever possible, we employ  $a_1 \text{andthen} a_2$  to combine actions: the declarative facet flows in parallel to  $a_1$  and  $a_2$  and the store and any temporaries thread from  $a_1$  to  $a_2$ . The language is an “andthen”-sequencing language. When we deviate from using **andthen** in the semantics, this indicates a feature deserving further study. Here are a few general points:

- The self-reference in  $\text{execute}[\text{while } E \text{ do } C]$  is understood as a lazy, infinite unfolding of the compound action, which has the usual least-fixed-point meaning in a partial ordering of partial, finite, and  $\omega$ -length phrases [6].
- Although the arity of **elaborate** states that an action may produce both a set of bindings (in  $\mathcal{D}$ ) and an altered store (in  $\mathcal{I}$ ), not all equations do so (e.g., **val** and **proc**). In these latter cases, the *R-strengthening* rule applies, passing the store through, unchanged.
- The *closure* defined in  $\text{elaborate}[\text{proc } I_1(I_2) = C]$  embeds the current bindings at the definition point. When applied, the closure’s action consumes the actual parameter, **it**, and the store at the point of call.

Figure 7 shows a derivation of the actions taken by a procedure call.

Here are the situations where combinators other than **andthen** appear:

- The **or** used in  $\text{evaluate}[N]$  gives opportunity to both its clauses to complete; at most one will do so. This is also true for  $\text{execute}[\text{while } E \text{ do } C]$ .
- **and** <sub>$\mathcal{FD}$</sub> **then** appears in situations (e.g.,  $\text{evaluate}[E_1 + E_2]$ ) where two arguments must be evaluated independently (“in parallel”) and supplied to a yielder, indicating that the transient values will be held for a longer extent

$\text{evaluate} : \text{Expression} \rightarrow \mathcal{DI} \rightarrow \mathcal{F}$ $\text{evaluate}[\![k]\!] = \text{give}_{\mathcal{F}} k$ $\text{evaluate}[\![E_1 + E_2]\!] = \begin{array}{l} (\text{evaluate } E_1 \text{ and}_{\mathcal{FD}} \text{then evaluate } E_2) \\ \text{andthen give}_{\mathcal{F}}(\text{add} (\text{isInt } \#1) (\text{isInt } \#2)) \end{array}$ $\text{evaluate}[\![N]\!] = \text{investigate } N \text{ andthen} \begin{array}{l} \text{lookup (isCell it)} \\ \text{or give}_{\mathcal{F}} (\text{isInt it}) \end{array}$  $\text{execute} : \text{Command} \rightarrow \mathcal{DI} \rightarrow \mathcal{I}$ $\text{execute}[\![N := E]\!] = \begin{array}{l} (\text{investigate } N \text{ and}_{\mathcal{FD}} \text{then evaluate } E) \\ \text{andthen update(isCell } \#1) \#2 \end{array}$ $\text{execute}[\![C_1; C_2]\!] = \text{execute } C_1 \text{ andthen execute } C_2$ $\quad \quad \quad \text{evaluate } E \text{ andthen} \\ \quad \quad \quad ((\text{give}_{\mathcal{F}}(\text{isZero it}) \text{ andthen complete})$ $\text{execute}[\![\text{while } E \text{ do } C]\!] = \begin{array}{l} \text{or} \\ \quad \quad \quad (\text{give}_{\mathcal{F}}(\text{isNonZero it}) \text{ andthen execute } C \\ \quad \quad \quad \text{andthen execute } [\![\text{while } E \text{ do } C]\!]) \end{array}$ $\text{execute}[\![D \text{ in } C]\!] = (\text{give}_{\mathcal{D}} \text{ currentbindings andthen elaborate } D) \text{ then execute } C$ $\text{execute}[\![\text{call } N(E)]\!] = \begin{array}{l} (\text{investigate } N \text{ and}_{\mathcal{FD}} \text{then evaluate } E) \\ \text{andthen exec(isClosure } \#1) \#2 \end{array}$  $\text{elaborate} : \text{Declaration} \rightarrow \mathcal{DI} \rightarrow \mathcal{DI}$ $\text{elaborate}[\![\text{val } I = E]\!] = \text{evaluate } E \text{ andthen give}_{\mathcal{D}} (\text{bind } I \text{ it})$ $\text{elaborate}[\![\text{var } I = E]\!] = \begin{array}{l} (\text{evaluate } E \text{ and}_{\mathcal{FD}} \text{then allocate}) \\ \text{andthen } (\text{give}_{\mathcal{D}} (\text{bind } I \#2) \text{ and}_{\mathcal{FD}} \text{then update } \#2 \#1) \end{array}$ $\text{elaborate}[\![\text{proc } I_1(I_2) = C]\!] = \text{give}_{\mathcal{D}}(\text{bind } I_1 \text{ closure})$ $\quad \quad \quad ((\text{give}_{\mathcal{D}} \text{ currentbindings} \\ \text{where } \text{closure} = \text{recabstract}_{\mathcal{D}} I_1 \text{ (currentbindings)} \quad \text{and}_{\mathcal{FD}} \text{then give}_{\mathcal{D}}(\text{bind } I_2 \text{ it})) \\ \quad \quad \quad \text{then execute } C)$ $\text{elaborate}[\![\text{module } I = D]\!] = \text{elaborate } D \text{ then give}_{\mathcal{D}}(\text{bind } I \text{ currentbindings})$ $\quad \quad \quad (\text{elaborate } D_1 \\ \quad \quad \quad \text{then } (\text{give}_{\mathcal{F}} \text{ currentbindings and give}_{\mathcal{D}} \text{ currentbindings}))$ $\text{elaborate}[\![D_1; D_2]\!] = \begin{array}{l} \text{andthen } ((\text{give}_{\mathcal{D}} \text{ currentbindings and}_{\mathcal{FD}} \text{then give}_{\mathcal{D}} \text{ it}) \\ \quad \quad \quad \text{then elaborate } D_2) \end{array}$  $\text{investigate} : \text{Name} \rightarrow \mathcal{D} \rightarrow \mathcal{F}$ $\text{investigate}[\![I]\!] = \text{give}_{\mathcal{F}}(\text{find } I)$ $\text{investigate}[\![N.I]\!] = \text{investigate } N \text{ then give}_{\mathcal{D}}(\text{isD it}) \text{ then give}_{\mathcal{F}}(\text{find } I)$
---

**Fig. 6.** Action equations

To make linear the big-step deductions that follow, we use this notation for subgoaling: For big-step rule,

$$\frac{\{\Gamma_i \vdash e_i \Rightarrow \tau_i\}_{i \in I} \quad \tau = f\{\tau_i\}_{i \in I}}{\Gamma \vdash \text{op}(e_i)_{i \in I} \Rightarrow \tau}$$

and goal,  $\Gamma \vdash \text{op}(e_i)_{i \in I} \Rightarrow \tau$ , we depict the subgoaling and computation of the result in this form:

$$:- \quad f\{\Gamma_i \vdash e_i \Rightarrow \tau_i\}_{i \in I} = \tau$$

Let  $\rho_{xp} = \{(x, \ell_0), (p, \text{closure}_p)\}$ ,  $\text{closure}_p = [\{(x, \ell_0)\}, p, [x := y]]_{\mathcal{D}}$ , and  $\sigma_x = [\ell_0 \mapsto 2]$ . This goal,

$$\rho_{xp}, \sigma_x \vdash \text{execute}[\text{call } p(3)] \Rightarrow \sigma_f$$

is solved for  $\sigma_f$  as follows:

$$\begin{aligned} &= \rho_{xp}, \sigma_x \vdash (\text{investigate } p \text{ and}_{FD} \text{then evaluate } 3) \\ &\quad \text{andthen exec(isClosure\#1) \#2} \Rightarrow \sigma_f \\ &:- \quad \rho_{xp}, \sigma_x \vdash (\text{investigate } p \text{ and}_{FD} \text{then evaluate } 3) \Rightarrow \tau_1, \sigma_x, \\ &\quad \tau_1, \rho_{xp}, \sigma_x \vdash \text{exec(isClosure\#1) \#2} \Rightarrow \sigma_2 = \sigma_f \end{aligned}$$

The first subgoal computes as follows:

$$\begin{aligned} &((\rho_{xp} \vdash \text{investigate } p \Rightarrow \text{closure}_p); (\rho_{xp} \vdash \text{evaluate } 3 \Rightarrow 3)) = \tau_1 \\ &= \langle \text{closure}_p, 3 \rangle = \tau_1 \end{aligned}$$

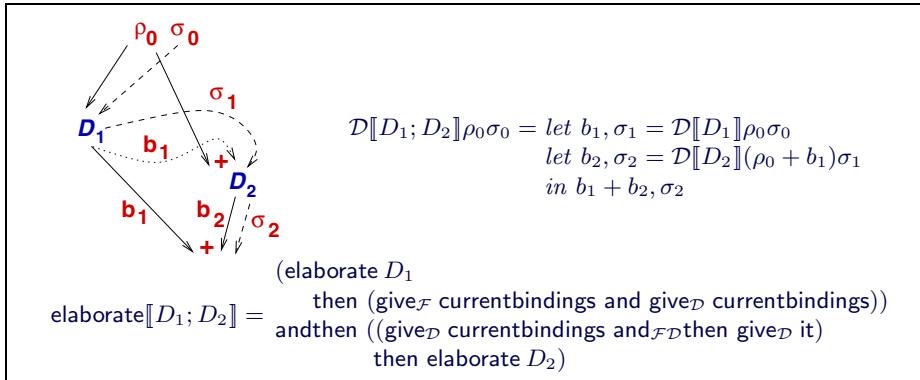
So, the second subgoal proceeds as follows:

$$\begin{aligned} &\langle \text{closure}_p, 3 \rangle, \rho_{xp}, \sigma_x \vdash \text{exec(isClosure\#1) \#2} \Rightarrow \sigma_f \\ &:- \quad \rho_{xp} + \{(y, 3)\}, \sigma_x \vdash \text{execute}[x := y] \Rightarrow \sigma_f \\ &= \rho_{xp} + \{(y, 3)\}, \sigma_x \vdash (\text{investigate } x \text{ and}_{FD} \text{then evaluate } y) \\ &\quad \text{andthen update(isCell \#1) \#2} \Rightarrow \sigma_f \\ &\dots = [\ell_0 \mapsto 3] = \sigma_f \end{aligned}$$

**Fig. 7.** Derivation of the actions defined by `call p(3)`

than usual. (An implementation might employ a stack to hold the longer-living transients.)

- `then` appears where the usual scoping is ignored (e.g., `execute[D in C]`). The strict sequential flow warns us that bindings are made locally and override the incoming scope. See in particular, `investigate[N.I]`, where the module (binding set) computed as  $N$ 's meaning replaces the current scope in determining  $I$ 's meaning.
- The binding flow for `elaborate[D1; D2]` is complex: The semantics assembles the bindings made by  $D_1$  and  $D_2$ , where  $D_1$  uses the entry scope, but  $D_2$  uses the scope made from the entry scope plus  $D_1$ 's bindings. This means  $D_1$ 's bindings are part of the output as well as part of the input to `elaborate[D2]`. For this reason, they are produced both as a declarative-facet value as well as a functional-facet value; see Figure 8.



**Fig. 8.** Facet flow of  $D_1; D_2$  and its denotational- and action-semantics codings

The denotational-semantics coding of the semantics, included in the Figure, states the correct distribution of bindings, but it ignores the language’s “semantic architecture” (i.e., the facets), which must accommodate  $D_1$ ’s binding flows.

Mosses named this complex flow pattern **elaborate  $D_1$  before elaborate  $D_2$**  [13]. Figure 9 shows a example derivation that uses the pattern.

As an exercise, one might rewrite the semantic equations so that the default combinator is **and<sub>FD</sub>then** (or, for that matter, **then** or **and**), to see what form of language results. (High-level declarative languages tend to be “**and**-languages,” and low-level imperative languages are “**then**-languages” where the store, symbol table, and temporary-value stack are passed sequentially.)

## 7 Pronoun Unambiguity and Mosses Abstraction

Although they are functions on facet values, the yielders **it** and **currentbindings** look like pronouns. For this reason, it is important these “pronouns” are understood unambiguously. Consider **elaborate**[ $D_1; D_2$ ] in Figure 8; there are three occurrences of pronoun **currentbindings**, but the first and second occurrences refer to the bindings generated from  $D_1$  and the third occurrence refers to the bindings incoming to  $D_1; D_2$ . We can repair the ambiguity with an abstraction form first proposed by Mosses [9], which we call the *Mosses abstraction*. Our version is an action of form,  $(p : G) \Rightarrow a$ , where pattern **p** predicts the shape of the incoming value from facet **G** to action **a**. Not a naive binder, pattern **p** defines named yielders that can be invoked within **a**. For example, the nested Mosses abstraction,

$$(\langle v, w \rangle : \mathcal{F}) \Rightarrow (\{(x, d)\} : \mathcal{D}) \Rightarrow \text{give}_{\mathcal{F}}(\text{add } w \ d)$$

asserts that the incoming functional-facet value is a sequence of at least two values and the incoming declarative-facet value holds at least a binding to **x**. The first pattern binds the name **v** to the yielder **#1** and binds the name **w** to the yielder **#2**; the second pattern binds the name **d** to the yielder, **find x**.

Let  $\rho_0 = \{\}$  and  $\sigma_0 = []$ . The goal

$$\rho_0, \sigma_0 \vdash \text{elaborate}[\text{var } x = 2; \text{ proc } p(y) = (x := y)] \Rightarrow \rho_{xp}, \sigma_x$$

is solved as follows:

$$\begin{aligned} & (\text{elaborate}[\text{var } x = 2]) \\ & \rho_0, \sigma_0 \vdash \text{then } (\text{give}_{\mathcal{F}} \text{ currentbindings and give}_{\mathcal{D}} \text{ currentbindings}) \Rightarrow \rho_{xp}, \sigma_x \\ & \text{andthen } ((\text{give}_{\mathcal{D}} \text{ currentbindings and }_{\mathcal{FD}} \text{ then give}_{\mathcal{D}} \text{ it}) \\ & \quad \text{then elaborate}[\text{proc } p(y) = (x := y)]) \\ & : - (\rho_0, \sigma_0 \vdash \text{elaborate}[\text{var } x = 2] \\ & \quad \text{then } (\text{give}_{\mathcal{F}} \text{ currentbindings and give}_{\mathcal{D}} \text{ currentbindings}) \Rightarrow \tau_1, \rho_x, \sigma_1) \\ & + (\tau_1, \rho_0, \sigma_1 \vdash (\text{give}_{\mathcal{D}} \text{ currentbindings and }_{\mathcal{FD}} \text{ then give}_{\mathcal{D}} \text{ it}) \\ & \quad \text{then elaborate}[\text{proc } p(y) = (x := y)] \Rightarrow \rho_p, \sigma_x) = \rho_{xp}, \sigma_x \end{aligned}$$

The first subgoal simplifies to

$$\begin{aligned} & = \{(x, \ell_0)\}, [\ell_0 \mapsto 2] \vdash \text{give}_{\mathcal{F}} \text{ currentbindings and give}_{\mathcal{D}} \text{ currentbindings} \Rightarrow \tau_1, \rho_x, \sigma_x \\ & = \langle \{(x, \ell_0)\}, \{(x, \ell_0)\}, [\ell_0 \mapsto 2] \rangle = \tau_1, \rho_x, \sigma_x \end{aligned}$$

Note how the binding,  $\{(x, \ell_0)\}$ , is copied to the functional facet as well as to the declarative facet. The overall denotation has progressed to

$$\begin{aligned} & (\{(x, \ell_0)\}) + \\ & \langle \{(x, \ell_0)\}, \rho_0, [\ell_0 \mapsto 2] \vdash (\text{give}_{\mathcal{D}} \text{ currentbindings and }_{\mathcal{FD}} \text{ then give}_{\mathcal{D}} \text{ it}) \\ & \quad \text{then elaborate}[\text{proc } p(y) = (x := y)] \Rightarrow \rho_p, \sigma_f \rangle = \rho_{xp}, \sigma_x \end{aligned}$$

The second subgoal proceeds as follows:

$$\begin{aligned} & \langle \{(x, \ell_0)\}, \rho_0, [\ell_0 \mapsto 2] \vdash (\text{give}_{\mathcal{D}} \text{ currentbindings and }_{\mathcal{FD}} \text{ then give}_{\mathcal{D}} \text{ it}) \\ & \quad \text{then elaborate}[\text{proc } p(y) = (x := y)] \Rightarrow \rho_p, \sigma_f \rangle \\ & : - (\{\} + \{(x, \ell_0)\} = \rho_x), (\rho_x, [\ell_0 \mapsto 2] \vdash \text{elaborate}[\text{proc } p(y) = (x := y)] \Rightarrow \rho_p, \sigma_f) \\ & = \{(x, \ell_0)\}, [\ell_0 \mapsto 2] \vdash \text{elaborate}[\text{proc } p(y) = (x := y)] \Rightarrow \rho_p, \sigma_f \\ & = \{(x, \ell_0)\}, [\ell_0 \mapsto 2] \vdash \text{give}_{\mathcal{D}}(\text{bind } p \text{ closure}_p) \Rightarrow \rho_p, \sigma_f \\ & \quad \text{where } \text{closure}_p = [\{(x, \ell_0)\}, p, [x := y]]_{\mathcal{D}} \\ & : - (\{(p, \text{closure}_p)\}, [\ell_0 \mapsto 2] = \rho_p, \sigma_f \end{aligned}$$

This makes the overall denotation equal

$$\{(x, \ell_0)\} + \{(p, \text{closure}_p)\}, [\ell_0 \mapsto 2] = \{(x, \ell_0), (p, \text{closure}_p)\}, [\ell_0 \mapsto 2] = \rho_{xp}, \sigma_x$$

**Fig. 9.** Actions taken by  $\text{var } x = 2; \text{ proc } p(y) = (x := y)$

Thus,  $\text{give}_{\mathcal{F}}(\text{add } w \ d)$  makes the same action as does  $\text{give}_{\mathcal{F}}(\text{add } \#2(\text{find } x))$ . A Mosses abstraction can be understood as a kind of “macro expansion,” much like traditional lambda notation macro-expands to De Bruijn notation. But there are crucial properties of Mosses abstractions that go beyond this simple analogy. To show this, we require a more formal development.<sup>2</sup>

<sup>2</sup> When he proposed the construction, Mosses stated, “The definition of ‘ $x \Rightarrow a_1$ ’ has been left informal, to avoid going into some technicalities.” [9]

A transient facet can be described by a pattern. The pattern we use for the functional facet,  $\mathcal{F}$ , is  $\langle v_i \rangle_{1 \leq i \leq k}$ , each  $v_i$  a name, representing a sequence of at least  $k$  values. For the declarative facet,  $\mathcal{D}$ , we use  $\text{rho}$  to denote the entire binding set and  $\{(x_i, d_i)\}_{1 \leq i \leq k}$ , each  $x_i$  an identifier and each  $d_i$  a name, to represent a binding set that has bindings for identifiers  $x_i$ .

Figure 10 defines how these patterns bind names to yielders. Yielders and actions are now evaluated with a *yielder environment*,  $\psi$ , a mapping of form,  $\text{Facet} \rightarrow \text{Identifier} \rightarrow \text{Yielder}$ . For the example Mosses abstraction seen earlier, action  $\text{give}_{\mathcal{F}}(\text{add } v \text{ } d)$  is interpreted with the yielder environment,  $[\mathcal{F} \mapsto [v \mapsto \#1, w \mapsto \#2], \mathcal{D} \mapsto [d \mapsto \text{find } x]]$ . Figure 10 shows and explains the derivation rule for yielder-name lookup, which consults  $\psi$  to extract and evaluate the corresponding yielder.

A Mosses abstraction is itself an action that operates with a yielder environment that is extended by the abstraction's pattern. *There can be at most one set of named yielders per facet*. Further, when a set of named yielders is generated for a facet, the default yielder for that same facet *cannot be used* — see the derivation rules for  $\#i$  and `currentbindings` in Figure 10. This removes pronoun/noun ambiguity in referencing values.

Additionally, a Mosses abstraction supports referential transparency. Within a Mosses-abstraction's body, *every reference to a yielder name evaluates to the same yielder which evaluates to the same value*. This crucial semantical property, which makes a Mosses abstraction behave like a lambda-abstraction, is ensured by the disciplined structure of the action-semantics combinator,  $a_1 \text{ andthen } a_2$ : if  $a_1$  generates output in facet  $\mathcal{F}$  that passes sequentially to  $a_2$ , then the  $\mathcal{F}$ -generated named yielders used by  $a_1$  are removed from  $a_2$ 's use — see Figure 10.

We can employ Mosses abstractions to clarify two definitions in Figure 6. First, the semantics of function definition now shows better how a closure uses its bindings and argument:

```
elaborate[proc I1(I2) = C] = (rho : D) => giveD(bind I1 closure)
where closure = recabstractD I1 rho
      (((arg) : F) =>
       (giveD rho andF,Dthen giveD(bind I2 arg))
       then execute C))
```

This macro-expands to the definition seen in Figure 6. (The proof depends on the big-step rule for `andF,Dthen`, which shows that the  $\mathcal{F}$ -value flows in parallel.)

Second, the multiple occurrences of `currentbindings` within the semantics of sequential declaration can be resolved unambiguously as

```
elaborate[D1; D2] = (rho0 : D) =>
  (elaborate D1 then (rho1 : D) => giveF rho1 and giveD rho1)
   andthen
   (((rho1) : F) => (giveD rho0 andF,Dthen giveD rho1) then elaborate D2))
```

Compare this semantics to the ones in Figure 8 — it is as readable as the denotational one but remains true to the underlying “semantic architecture.”

Each facet pattern generates a *yielder environment* of arity,  $\text{Facet} \rightarrow \text{Identifier} \rightarrow \text{Yielder}$ :

$$\begin{aligned}\llbracket \langle v_i \rangle_{1 \leq i \leq k} : \mathcal{F} \rrbracket &= [\mathcal{F} \mapsto [v_i \mapsto \#i]_{1 \leq i \leq k}] \\ \llbracket \text{rho} : \mathcal{D} \rrbracket &= [\mathcal{D} \mapsto [\text{rho} \mapsto \text{currentbindings}]] \\ \llbracket \{\langle x_i, d_i \rangle\}_{1 \leq i \leq k} : \mathcal{D} \rrbracket &= [\mathcal{D} \mapsto [d_i \mapsto \text{find } x_i]_{1 \leq i \leq k}]\end{aligned}$$

Let  $\psi$  be a yielder environment. A yielder sequent now has form,  $\Gamma \vdash_\psi y : \tau$ ; the  $\psi$  annotations are uniformly added to the sequents in the rules of Figure 1. The rule for evaluating a yielder name,  $n$ , defined from the pattern for facet  $G$ , goes as follows:

$$\frac{G \in \text{domain}(\psi) \quad \psi(G)(n) = y \quad \Gamma \vdash y : \tau}{\Gamma \vdash_\psi (n : G) : \tau}$$

The default yielders for  $\mathcal{F}$  and  $\mathcal{D}$  are “disabled” when a yielder environment already exists for the facet:

$$\frac{\mathcal{F} \notin \text{domain}(\psi) \quad 1 \leq i \leq n}{(\tau_1, \dots, \tau_n) \vdash_\psi \#i : \tau_i} \quad \frac{\mathcal{D} \notin \text{domain}(\psi)}{\rho \vdash_\psi \text{currentbindings} : \rho}$$

An action sequent has form,  $\Gamma \vdash_\psi a \Rightarrow \Delta$ . The  $\psi$  annotations are uniformly added to the sequents in the rules of Figure 2. The new rule for Mosses abstraction reads as follows:

$$\frac{\Gamma \vdash_{\psi + \llbracket p \rrbracket} a \Rightarrow \Delta}{\Gamma \vdash_\psi (p \Rightarrow a) \Rightarrow \Delta}$$

The  $+$  denotes function override.

When an  $\mathcal{F}$ -value flows sequentially from action  $a_1$  to  $a_2$ , the  $\psi(\mathcal{F})$ -part of  $\psi$  must be removed from  $a_2$ 's use. This is enforced by the revised rule for `andthen`:

$$\frac{\Gamma \vdash_\psi a_1 \Rightarrow \Delta_1 \quad (\Gamma \downarrow_G) \cup (\Delta_1 \downarrow_{\sim G}) \vdash_{\psi \downarrow G} a_2 \Rightarrow \Delta_2}{\Gamma \vdash_\psi a_1 \text{ and}_G \text{then } a_2 \Rightarrow \Delta_1 \downarrow_G \circ \Delta_2}$$

where  $\psi \downarrow G$  denotes  $\psi$  restricted to argument(s)  $G$  only.

Finally, the yielder environment for a closure is restricted to the facets embedded within the closure:

$$\frac{\begin{array}{c} \Gamma \vdash_\psi y : \Delta \\ \Gamma \vdash_\psi \text{recabstract}_G y I a : [\Delta \downarrow_G, I, a]_{G, \psi \downarrow G} \\ \Gamma_1 \vdash_\psi y_1 : [\Delta, I, a]_{G, \psi'} \\ \Gamma_2 \vdash_\psi y_2 : \tau \\ \Gamma = \Gamma_1 \cup \Gamma_2 \end{array}}{\Gamma \vdash_\psi \text{exec } y_1 y_2 \Rightarrow \Sigma}$$

**Fig. 10.** Semantics of Mosses abstractions

## 8 From Action Equations to Big-Step Semantics

Since the yielders and actions are defined by big-step semantic rules, one can map the action equations themselves into big-step-rule format by applying partial evaluation [1,8]. The idea is that a valuation function of arity,  $\text{interp} : \text{PhraseForm} \rightarrow \mathcal{F}_1 \cdots \mathcal{F}_m \rightarrow \mathcal{G}_1 \cdots \mathcal{G}_n$ , suggests a big-step sequent of form,  $f_1 \circ \cdots \circ f_m \vdash P \Rightarrow g_1 \circ \cdots \circ g_n$ . An action equation for a phrase,  $\text{cons } P_1 \cdots P_p$ , relies on the actions denoted by the  $\text{interp } P_i$ s to compute  $\text{interp}[\text{cons } P_1 \cdots P_p]$ . The corresponding big-step rule uses the sequent forms for each  $P_i$  as antecedents for the consequent sequent,  $f_1 \circ \cdots \circ f_m \vdash \text{cons } P_1 \cdots P_p \Rightarrow g_1 \circ \cdots \circ g_n$ .

The translation from action equations to big-step rules is a mechanical process, where the big-step rules for **or**, **andthen**, and the primitive actions and yielders are elaborated to expose the argument-passing flows of temporaries,  $\tau$ , binding sets,  $\rho$ , and store,  $\sigma$ . Figure 11 shows representative translations from Figure 6. Of the examples shown above, only the rule for **call N(E)** requires mild reformatting to match its counterpart in Figure 6, the issue being the binding of actual to formal parameter.

$\rho \circ \sigma \vdash \text{evaluate } E_1 \Rightarrow \tau_1$	$\rho \circ \sigma \vdash \text{evaluate } E_1 \Rightarrow \tau_2$	$\tau_1 \leq \text{Int}$	$\tau_2 \leq \text{Int}$
			$\tau_3 = \text{add}(\tau_1, \tau_2)$
$\rho, \sigma \vdash \text{evaluate}[E_1 + E_2] \Rightarrow \tau_3$			
<hr/>			
$\rho \vdash \text{investigate } N \Rightarrow \tau$			
$\tau \leq \text{Cell}$			
$\sigma(\tau_1) = \tau_2$			
$\rho \circ \sigma \vdash \text{evaluate } N \Rightarrow \tau_2$			
<hr/>			
$\rho \circ \sigma \vdash \text{investigate } N \Rightarrow \tau_1$			
$\tau_1 \leq \text{Cell}$			
$\rho \circ \sigma \vdash \text{evaluate } E \Rightarrow \tau_2$			
$\sigma_1 = \sigma[\tau_1 \mapsto \tau_2]$			
$\rho \circ \sigma \vdash \text{execute}[N := E] \Rightarrow \sigma_1$			
<hr/>			
$\rho \circ \sigma \vdash \text{execute } C_1 \Rightarrow \sigma_1$			
$\rho \circ \sigma_1 \vdash \text{execute } C_2 \Rightarrow \sigma_2$			
$\rho \circ \sigma \vdash \text{execute}[C_1; C_2] \Rightarrow \sigma_2$			
<hr/>			
$\rho \circ \sigma \vdash \text{evaluate } E \Rightarrow \tau$			
$\tau \leq \text{NonZero}$			
$\rho \circ \sigma \vdash \text{execute } C \Rightarrow \sigma_1$			
$\rho \circ \sigma_1 \vdash \text{execute}[\text{while } E \text{ do } C] \Rightarrow \sigma_2$			
$\rho \circ \sigma \vdash \text{execute}[\text{while } E \text{ do } C] \Rightarrow \sigma_2$			
<hr/>			
$\rho \circ \sigma \vdash \text{investigate } N \Rightarrow \tau_1$			
$\tau_1 \leq \text{Closure}$			
$\rho \circ \sigma \vdash \text{evaluate } E \Rightarrow \tau_2$			
$\rho_1 + \{(I_1, \tau_1), (I_2, \tau_2)\} \circ \sigma \vdash \text{execute } C \Rightarrow \sigma_1$			
$\rho \circ \sigma \vdash \text{execute}[\text{call } N(E)] \Rightarrow \sigma_1$			
<hr/>			
$\rho \circ \sigma \vdash \text{evaluate } E \Rightarrow \tau$			
$c \notin \text{domain}(\sigma)$			
$\rho_1 = \{(I, c)\}$			
$\sigma_1 = \sigma[c \mapsto \tau]$			
$\rho \circ \sigma \vdash \text{elaborate}[\text{var } I = E] \Rightarrow \rho_1 \circ \sigma_1$			
<hr/>			
$\rho \circ \sigma \vdash \text{elaborate } D_1 \Rightarrow \rho_1 \circ \sigma_1$			
$\rho \circ \rho_1 \circ \sigma_1 \vdash \text{elaborate } D_2 \Rightarrow \rho_2 \circ \sigma_2$			
$\rho_3 = \rho_1 \circ \rho_2$			
$\rho \circ \sigma \vdash \text{elaborate}[D_1; D_2] \Rightarrow \rho_3 \circ \sigma_2$			

**Fig. 11.** Selected big-step rules derived from action equations

Figures 6 and 11 have the same information content, but Figure 6 is higher-level in its presentation of value flows, whereas Figure 11 makes explicit the connections and compositions. A reader familiar with big-step semantics may prefer the latter, but the explicit detail obscures the fundamental **andthen** facet flows that give the language its character. And this was indeed the issue that motivated Mosses to develop action semantics in the first place.

## 9 From Action Equations to Denotational Semantics

Mosses did not intend to erase from memory Scott-Strachey denotational semantics [7,14,19], but he desired a methodology and notation that matched more closely a programmer’s and a language designer’s intuitions and was less sensitive to modelling issues (e.g., direct versus continuation semantics). Action semantics lives at a higher level of abstraction than does denotational semantics, and it is a routine but enlightening exercise to interpret its facets, yielders, and actions with Scott-domains and continuous functions:

- Each facet is mapped to a Strachey-style “characteristic domain.”
- Each yielder is mapped to a continuous function on the characteristic domains such that the big-step rules in Figure 1 are proved sound, where  $\Gamma \vdash y : \Delta$  is interpreted as  $\llbracket y \rrbracket(\gamma) = \delta$ , such that  $\llbracket y \rrbracket$  is the continuous function and  $\gamma$  and  $\delta$  are the Scott-domain values interpreted from  $\Gamma$  and  $\Delta$ .
- Each action is mapped to a (higher-order) continuous function in a similar manner so that the rules in Figure 2 are proved sound. When the yielders compute on transients only and actions compute on persistent values, the interpretation into denotational semantics gives a “two-level” denotational semantics as developed by Nielson and Nielson [17].
- The action equations are treated as valuation functions, and a least-fixed-point interpretation is taken of self-references.

Once again, the information in the action-equation version and the denotational semantics version is “the same,” but the former presents and emphasizes language design concepts more directly.

Related to this exercise is the relationship between direct-style and continuation-style semantics. If our example language contained jumps or fork-join constructions, the denotational-semantics domains would change as would the interpretation of the **andthen** combinator, into a tail-recursive form, *à la a<sub>1</sub> andthen \_*, so that the continuation from  $a_1$  could be kept or discarded as directed. The action equations themselves remain the same.

## 10 Subsorts within the Facets

The relation,  $\leq$ , has been used to indicate data-type membership, e.g.,  $3 \leq \text{Int}$ . Mosses defined *unified algebra* [12] to state membership properties that go beyond the usual judgements.

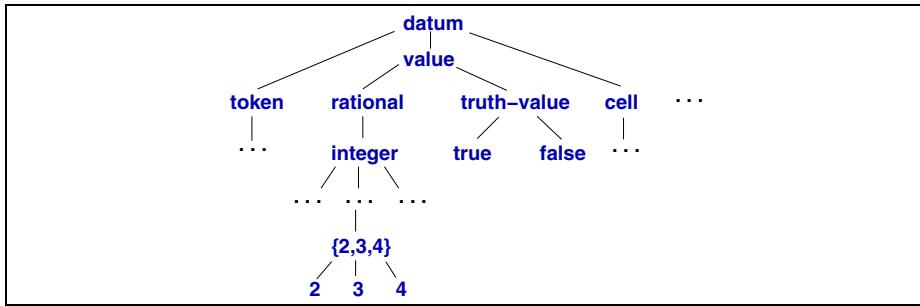


Fig. 12. Sort structure for functional facet

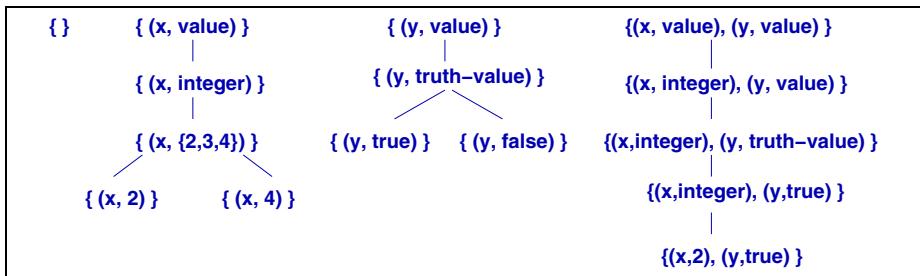


Fig. 13. Sort structure for declarative facet

Figure 12 portrays a subsort relationship, a partial ordering, for a unified algebra that lists the sorts of values that can be used with the functional facet. The diagram asserts that  $\text{integer} \leq \text{rational} \leq \text{datum}$ , etc. — an implicit subset ordering applies. Even finite sets of values ( $\{2, 3, 4\}$ ) and singletons ( $2$ ) are sorts. Of course, one does not implement all the sort names, but they serve as a useful definitional tool. Each sort name is interpreted by a carrier of values that belong to the sort, e.g.,  $\text{integer}$  is interpreted by  $\{\dots, -1, 0, 1, 2, \dots\}$ , and  $\{2, 3, 4\}$  is interpreted by  $\{2, 3, 4\}$ . Sequences of functional-facet values are ordered pointwise.

The declarative facet's sorts can be portrayed as seen in Figure 13. There is a pointwise ordering, based on the identifiers that are named in the sorts:  $\rho_1 \leq \rho_2$  iff the identifiers named in  $\rho_1$  equal the ones named in  $\rho_2$  and for every  $(I, \tau_1)$  in  $\rho_1$  and  $(I, \tau_2)$  in  $\rho_2$ ,  $\tau_1 \leq \tau_2$ . A sort is interpreted by those sets of pairs that have exactly the bindings stated in the sort name, e.g.,  $\{(x, \text{integer}), (y, \{2, 3, 4\})\}$  is interpreted by  $\{\{(x, m), (y, n)\} \mid m \leq \text{integer} \text{ and } n \leq \{2, 3, 4\}\}$ .

The imperative facet is organized similarly to the declarative facet: its sorts are finite maps from Cell names to subsorts of datum such that  $\sigma_1 \leq \sigma_2$  iff  $\text{domain}(\sigma_1) = \text{domain}(\sigma_2)$  and for every  $c \in \text{domain}(\sigma_1)$ ,  $\sigma_1(c) \leq \sigma_2(c)$ .

Strictly speaking, subsorting does not extend to compound-facet values, but we will write  $\Delta \leq \Gamma$  for compound facet values to assert that  $\Delta = \{f_1, \dots, f_m\}$ ,  $\Gamma = \{f'_1, \dots, f'_m\}$  and  $f_i \leq f'_i$  for all  $i \in 1..m$  in each respective facet,  $\mathcal{F}_i$ .

The sorting hierarchies suggest that yielders and actions can compute on sort names as well as individual values. For example, all these derivations are meaningful:

$$\begin{aligned}
 & \vdash \text{give } 2 \Rightarrow 2 \\
 & \vdash \text{give } 2 \Rightarrow \text{integer} \\
 & \vdash \text{give } 2 \Rightarrow \text{datum} \\
 \{ \langle 2 \rangle, \{(x, 3)\} \} & \vdash \text{give add(isInteger it, isInteger(find } x)) \Rightarrow 5 \\
 \{ \langle 2 \rangle, \{(x, 3)\} \} & \vdash \text{give add(isInteger it, isInteger(find } x)) \Rightarrow \text{integer} \\
 \{ \langle 2 \rangle, \{(x, \text{integer})\} \} & \vdash \text{give add(isInteger it, isInteger(find } x)) \Rightarrow \text{integer} \\
 \{ \langle \text{integer} \rangle, \{(x, \text{integer})\} \} & \vdash \text{give add(isInteger it, isInteger(find } x)) \Rightarrow \text{integer}
 \end{aligned}$$

The derivations justify type checking and abstract interpretation upon the semantics definition.<sup>3</sup>

The above assertions are connected by this weakening rule for sorting, which weakens information within individual facets:

$$\frac{\Gamma_1 \leq \Gamma_2 \quad \Gamma_2 \vdash a \Rightarrow \Delta_2 \quad \Delta_2 \leq \Delta_1}{\Gamma_1 \vdash a \Rightarrow \Delta_1}$$

The rule complements the two existing weakening and strengthening rules, which weaken the facet structures themselves.

## 11 Partial Evaluation of Action Semantics

As in the derivation examples in the previous section, a yielder may have many possible input-output sort pairs. However, for input context  $\Gamma$ , there is a least output sort  $\Delta$  such that  $\Gamma \vdash y : \Delta$  holds. This is called the *least sorting property*. For example, a yielder 2 can have output sorts datum, value, integer, {2,3,4}, 2, etc., in any input context, but the least sort among them is 2. Similarly, a yielder add (isInteger it) (isInteger (find x)) has the least output sort integer in the context, 2, {(x,integer)}. The sort checking rule for calculating a least sort for integer addition operation can be defined as follows:

$$\frac{\Gamma \vdash y_1 : \tau_1 \quad \Delta \vdash y_2 : \tau_2 \quad \tau_1 \leq \text{integer} \quad \tau_2 \leq \text{integer}}{\Gamma \cup \Delta \vdash \text{add } y_1 \ y_2 : \text{add}(\tau_1, \tau_2)}$$

Other rules for yielders in Figure 1 can be used for sort checking without modification, along with the weakening rule just introduced:

$$\frac{\Gamma_1 \leq \Gamma_2 \quad \Gamma_2 \vdash y : \Delta_2 \quad \Delta_2 \leq \Delta_1}{\Gamma_1 \vdash y : \Delta_1}$$

The rules for sort consistency check upon actions can also be defined in big-step deduction style. Read  $\Gamma \vdash a \Rightarrow \Delta$  as asserting that action  $a$  receives sorts of facets  $\Gamma$  and produces sorts of facets  $\Delta$ . Note that the imperative facet is excluded from  $\Gamma$  and  $\Delta$  since sort checking occurs before run-time. The sort checking rules for imperative-facet actions are defined as follows:

<sup>3</sup> In the example, we assume that the operation, *add*, is extended monotonically to operate on all sorts within the functional facet. This makes all yielders and actions behave monotonically as well.

$$\frac{\Gamma \vdash y : c \quad c \leq \text{integer-cell}}{\Gamma \vdash \text{lookup } y \Rightarrow \text{integer}}$$

$$\frac{\Gamma_1 \vdash y_1 : c \quad c \leq \text{integer-cell} \quad \Gamma_2 \vdash y_2 : \tau \quad \tau \leq \text{integer}}{\Gamma_1 \cup \Gamma_2 \vdash \text{update } y_1 \ y_2 \Rightarrow \text{completing}}$$

$$\vdash \text{allocate} \Rightarrow \text{integer-cell}$$

Sort consistency checking between closure and its arguments is done at application time. The rules are identical to those in Figure 2.

Sorts can be distinguished according to binding times — compile-time sorts and run-time sorts [3,5]. Individual sorts, such as 1, true, etc., are known constants, and thus static sorts. All other sorts, including integer, cell, etc., are treated as dynamic sorts because their values are not known. Yielders and actions taking static sorts can be processed at compile-time, reducing the run-time computation overhead. For example, consider an action, give 2 andthen give (add it (find x)). The left subaction, give 2, is statically computable, and passes its output value to the yielder it in the right subaction. The yielder it then consumes the value, and then the whole action is semantically identical to give 2 andthen give (add 2 (find x)). Since the life of the left subaction is over, the action is safely reduced to give (add 2 (find x)). If the given context to this action is {(x,3)}, find x becomes 3, and then the yielder add 2 3 is further computed to 5. On the other hand, if the given context is {(x,integer)}, no more computation is possible and the action remains as it is. This transformation is partial evaluation of the actions.

During partial evaluation, since each yielder either gives computed sorts or reconstructs yielder code, rules for partial evaluation have to carry around both facets and reconstructed residual code. Read  $\Gamma, \kappa_i \vdash y : \Delta, \kappa_o$  as asserting that yielder  $y$  consumes facets  $\Gamma$  and residual code  $\kappa_i$ , and produces facets  $\Delta$  and residual code  $\kappa_o$ . If the output sort  $\Delta$  is static, then code reconstruction is not necessary and thus  $\kappa_o = \emptyset$ , indicating no code. Otherwise, the residual code is reconstructed.

Checking whether or not a yielder output is static can be done by examining its output sort. The following function *static* determines if the given functional-facet sort is static.

$$\begin{aligned}
 \text{static}(\langle \tau \rangle) &= \text{if } \tau \text{ is an individual value then true else false} \\
 \text{static}(\langle \tau_1, \dots, \tau_n \rangle) &= \text{static}(\tau_1) \wedge \dots \wedge \text{static}(\tau_n) \\
 \text{static}([\tau, I, a]) &= \text{false} \\
 \text{static}(\{(I_1, \tau_1), \dots, (I_n, \tau_n)\}) &= \text{static}(\tau_1) \wedge \dots \wedge \text{static}(\tau_n)
 \end{aligned}$$

A yielder output is also static when the emitted residual code is  $\emptyset$ .

Figure 14 presents a sample collection of rules for partial evaluation for yielders. Primitive constant yielder is always static, thus emits no residual code,  $\emptyset$ . If two argument yielders of addition operation are both static, they are evaluated and added to give a static output, while emitting no code. Otherwise, after two argument yielders are partially evaluated, the whole yielder code is reconstructed. If the output of declarative-facet yielder is static, no code is emitted. However, the code is reconstructed otherwise.

*Functional-facet yielders:*

primitive constant:  $\vdash k : k, \emptyset$

n-ary operation (e.g., addition):

$$\frac{\Gamma, \kappa \vdash y_1 : \tau_1, \kappa_1 \quad \Delta, \kappa \vdash y_2 : \tau_2, \kappa_2}{\begin{array}{c} \text{case } \kappa_1, \kappa_2 \text{ of} \\ \emptyset, \emptyset \rightarrow add(\tau_1, \tau_2), \emptyset \\ \emptyset, - \rightarrow \tau_1 \sqcup \tau_2, [add [\tau_1] \kappa_2] \\ -, \emptyset \rightarrow \tau_1 \sqcup \tau_2, [add \kappa_1 [\tau_2]] \\ -, - \rightarrow \tau_1 \sqcup \tau_2, [add \kappa_1 \kappa_2] \end{array}}$$

indexing:  $\frac{1 \leq i \leq m}{\langle \tau_1, \dots, \tau_m \rangle, \langle \kappa_1, \dots, \kappa_m \rangle \vdash \#i : \tau_i, \kappa_i}$  Note: it abbreviates #1

sort filtering:  $\frac{\Gamma, \kappa \vdash y : \Delta, \kappa' \quad \Delta \leq T}{\Gamma, \kappa \vdash \text{is}T y : \Delta, \text{if } \kappa' = \emptyset \text{ then } \emptyset \text{ else } [\text{is}T \kappa']}$

*Declarative-facet yielders:*

binding lookup:  $\frac{(l, \tau) \in \rho}{\rho, \kappa \vdash \text{find } l : \tau, \text{if static}(\tau) \text{ then } \emptyset \text{ else } [\text{find } l]}$

binding creation:  $\frac{\Gamma, \kappa \vdash \text{bind } l y : \{(l, \tau)\}, \text{if } \kappa' = \emptyset \text{ then } \emptyset \text{ else } [\text{bind } l \kappa']}$

binding copy:  $\frac{\rho, \kappa \vdash \text{currentbindings} : \rho, \text{if } \kappa' = \emptyset \text{ then } \emptyset \text{ else } [\text{currentbindings}]}{}$

**Fig. 14.** Partial evaluation for yielders

*Structural actions:*

$\frac{\Gamma, \kappa \vdash y : \Delta, \kappa'}{\Gamma, \kappa \vdash \text{give}_G y \Rightarrow \Delta, \text{if } \kappa' = \emptyset \text{ then } \emptyset \text{ else } [\text{give}_G \kappa']}$

$\vdash \text{complete} \Rightarrow \text{completing}, [\text{complete}]$

*Imperative-facet actions:*

$\frac{\Gamma, \kappa \vdash y : s\text{-cell}, \kappa'}{\Gamma, \kappa \vdash \text{lookup } y \Rightarrow s, [\text{lookup } \kappa']}$

$\frac{\Gamma_1, \kappa \vdash y_1 : \tau_1\text{-cell}, \kappa_1 \quad \Gamma_2, \kappa \vdash y_2 : \tau_2, \kappa_2 \quad \tau_2 \leq \tau_1}{\Gamma_1 \cup \Gamma_2, \kappa \vdash \text{update } y_1 y_2 \Rightarrow \text{completing}, [\text{update } \kappa_1 \kappa_2]}$

$\vdash \text{allocate}_s \Rightarrow s\text{-cell}, [\text{allocate}_s]$

**Fig. 15.** partial evaluation for actions

Figure 15 presents a sample action set, whose partial-evaluation behaviors are defined with big-step deduction rules. Read  $\Gamma, \kappa_i \vdash a \Rightarrow \Delta, \kappa_o$  as asserting that action  $a$  receives facets  $\Gamma$  and a residual code  $\kappa_i$ , and produces facets  $\Delta$  and a residual code  $\kappa_o$ . A structural action such as give  $y$  either gives the evaluated results or reconstructs a residual code depending on its binding time. Imperative actions are all reconstructed at partial-evaluation time, but their yielder constituents are evaluated when possible. Since the termination is not guaranteed, the body of a self-referencing closure is not partially evaluated.

For action combinators,  $\text{and}_G \text{then}$ , partial evaluation is defined differently depending on the facet flows. When  $G = \emptyset$ , the combinator is essentially the same as  $\text{then}$ , and its partial evaluation can be defined as follows:

$$\frac{\Gamma, \kappa \vdash a_1 \Rightarrow \Delta, \kappa_1 \quad \Delta, \kappa_1 \vdash a_2 \Rightarrow \Sigma, \kappa_2}{\Gamma, \kappa \vdash a_1 \text{ then } a_2 \Rightarrow \Sigma, \text{ if } \kappa_1 = \emptyset \text{ then } \kappa_2 \text{ else } [\kappa_1 \text{ then } \kappa_2]}$$

In this case, when the left subaction is static, the whole action can be reduced to its right subaction. The partial evaluation of the  $\text{and}_G \text{then}$  combinator when  $G \neq \emptyset$  is defined differently as follows:

$$\frac{\begin{array}{c} \Gamma, \kappa \vdash a_1 \Rightarrow \Delta_1, \kappa_1 \quad (\Gamma \downarrow_G) \cup (\Delta_1 \downarrow_{\sim G}), \kappa \vdash a_2 \Rightarrow \Delta_2, \kappa_2 \\ \hline \text{case } \kappa_1, \kappa_2 \text{ of} \\ \quad \emptyset, \emptyset \rightarrow \emptyset \\ \quad \emptyset, - \rightarrow [[\Delta_1]] \text{ and}_G \text{then } \kappa_2 \\ \quad -, \emptyset \rightarrow [\kappa_1 \text{ and}_G \text{then } [\Delta_2]] \\ \quad -, - \rightarrow [\kappa_1 \text{ and}_G \text{then } \kappa_2] \end{array}}{\Gamma, \kappa \vdash a_1 \text{ and}_G \text{then } a_2 \Rightarrow \Delta_1 \downarrow_G \circ \Delta_2, \quad \emptyset, - \rightarrow [[\Delta_1]] \text{ and}_G \text{then } \kappa_2 \\ \quad -, \emptyset \rightarrow [\kappa_1 \text{ and}_G \text{then } [\Delta_2]] \\ \quad -, - \rightarrow [\kappa_1 \text{ and}_G \text{then } \kappa_2]}$$

## 12 Conclusion

Action semantics was an influential experiment in programming-language design and engineering. Its success rests on its relatively high level of abstraction and its sensitivity to a language’s “semantic architecture,” as expressed by facets. Action semantics readily maps to operational and denotational semantics definitions, giving a good entry point into language-definition methodology.

By presenting a naive formulation, based on two combinators, a weakening rule, and a strengthening rule, the present paper has attempted to expose action semantics’s personality and emphasize its strengths.

## In appreciation of Peter D. Mosses

Peter Mosses has made significant impact on the programming languages community, and he made significant impact on the authors of this paper as well.

*David Schmidt:* I met Peter in Aarhus in 1979. I was impressed by his thesis work, his Semantics Implementation System, and his ability to go to the core of an issue. (One example: After viewing a presentation on a dubiously complex

programming technique, Peter contributed, “As Strachey might say, first get it working correctly, then get it working fast!”)

Peter answered many of my beginner’s questions about denotational semantics, and his work on binding algebras showed me that language semantics was more than Scott-domains and lambda calculus — semantics itself should have structure. This insight, plus Peter’s remark to me in 1982, that semantic domains need not necessarily possess  $\perp$ -elements, gave me all I needed to write the key chapters of my *Denotational Semantics* text [18].

Peter’s research on action semantics forms the most profound body of knowledge on programming-language principles I have encountered, and I have enjoyed studying and applying this material over the decades. I deeply appreciate Peter’s insights, his perseverance, and his friendship.

*Kyung-Goo Doh:* It was the paper, “Abstract semantic algebras!” [10], introduced to me by David Schmidt in 1990, that impressed me and guided me into the Peter Mosses’s world of programming language semantics. It did not take a long time for me to choose the subject as my Ph.D. research topic. Since then, Peter has been a good mentor to me on numerous occasions through personal communications, research collaborations, and published papers. Peter’s outstanding works in programming-language semantics hugely influenced me and my students on understanding the principles of programming languages. Peter assured me that it would be possible to have a useful semantics formalism for realistic programming languages just like the syntax counterpart, BNF. I admire Peter’s enduring body of research works, and I cordially thank him for his support and friendship.

Finally, this paper is a significantly expanded and revised version of research presented at the First Workshop on Action Semantics, Edinburgh, 1994 [4]. The authors thank Peter Mosses for organizing the workshop and inviting both of us to attend.

## References

1. Doh, K.-G.: Action transformation by partial evaluation. In: PEPM 1995, pp. 230–240. ACM Press, New York (1995)
2. Doh, K.-G., Mosses, P.D.: Composing programming languages by combining action-semantics modules. Sci. Computer Prog. 47, 3–36 (2003)
3. Doh, K.-G., Schmidt, D.A.: Extraction of strong typing laws from action semantics definitions. In: Krieg-Brückner, B. (ed.) ESOP 1992. LNCS, vol. 582, pp. 151–166. Springer, Heidelberg (1992)
4. Doh, K.-G., Schmidt, D.A.: The facets of action semantics: some principles and applications. In: Workshop on Action Semantics, pp. 1–15. Univ. of Aarhus, BRICS NS-94-1 (1994)
5. Doh, K.-G., Schmidt, D.A.: Action semantics-directed prototyping. Computer Languages 19, 213–233 (1993)
6. Guessarian, I.: Algebraic Semantics. In: Guessarian, I. (ed.) Algebraic Semantics. LNCS, vol. 99, Springer, Heidelberg (1981)

7. Gunter, C., Scott, D.S.: Semantic domains. In: *Handbook of Theoretical Computer Science*, vol. B, pp. 633–674. MIT Press, Cambridge (1991)
8. Jones, N.D., Gomard, C., Sestoft, P.: *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs (1993)
9. Mosses, P.D.: A semantic algebra for binding constructs. In: Díaz, J., Ramos, I. (eds.) *Formalization of Programming Concepts*. LNCS, vol. 107, pp. 408–418. Springer, Heidelberg (1981)
10. Mosses, P.D.: Abstract semantic algebras! In: *Formal Description of Programming Concepts II*, Proceedings of the IFIP TC2 Working Conference, Garmisch-Partenkirchen, 1982, pp. 45–72. IFIP, North-Holland, Amsterdam (1983)
11. Mosses, P.D.: A basic abstract semantic algebra. In: Plotkin, G., MacQueen, D.B., Kahn, G. (eds.) *Semantics of Data Types 1984*. LNCS, vol. 173, pp. 87–107. Springer, Heidelberg (1984)
12. Mosses, P.D.: Unified algebras and action semantics. In: Cori, R., Monien, B. (eds.) *STACS 1989*. LNCS, vol. 349, Springer, Heidelberg (1989)
13. Mosses, P.D.: *Action Semantics*. Cambridge Tracts in Theoretical Computer Science, vol. 26. Cambridge University Press, Cambridge (1992)
14. Mosses, P.D.: Denotational semantics. In: van Leeuwen, J. (ed.) *Handbook of Theoretical Computer Science*. Formal Models and Semantics (B), vol. B, pp. 575–631. MIT Press, Cambridge (1990)
15. Mosses, P.D.: Theory and practice of action semantics. In: Penczek, W., Szałas, A. (eds.) *MFCS 1996*. LNCS, vol. 1113, pp. 37–61. Springer, Heidelberg (1996)
16. Mosses, P.D., Watt, D.A.: The use of action semantics. In: Wirsing, M. (ed.) *Formal Description of Programming Concepts III*, Proc. IFIP TC2 Working Conference, Gl. Avernaes, 1986, IFIP, North-Holland (1987)
17. Nielson, F., Nielson, H.R.: *Two-Level Functional Languages*. Cambridge University Press, Cambridge (1992)
18. Schmidt, D.A.: *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc. (1986)
19. Scott, D.S., Strachey, C.: Toward a mathematical semantics for computer languages. In: Fox, J. (ed.) *Proceedings of Symposium on Computers and Automata*, Polytechnic Institute of Brooklyn. Microwave Research Institute Symposia Series, vol. 21, pp. 19–46 (1971)
20. Stoy, J.E.: *Denotational Semantics*. MIT Press, Cambridge (1977)
21. Strachey, C.: The varieties of programming language. Technical Report PRG-10, Prog. Research Group, Oxford University (1973)
22. Watt, D.A.: An action semantics of standard ML. In: Main, M.G., Mislove, M.W., Melton, A.C., Schmidt, D. (eds.) *MFPS 1987*. LNCS, vol. 298, pp. 572–598. Springer, Heidelberg (1988)

# Converting between Combinatory Reduction Systems and Big Step Semantics

Hanne Gottliebsen<sup>1,\*</sup> and Kristoffer H. Rose<sup>2</sup>

<sup>1</sup> Brorsonsgade 8, 1.th, 1624 Copenhagen V, Denmark

[hanne@gottliebsen.dk](mailto:hanne@gottliebsen.dk)

<sup>2</sup> IBM Thomas J. Watson Research Center

[krisrose@us.ibm.com](mailto:krisrose@us.ibm.com)

**Abstract.** We make a connection between higher-order rewriting in the form of combinatory reduction systems (CRS) and logic-based operational semantics in the form of big step semantic (BSS) specifications. We show how sets of CRS rewrite rules can be encoded as BSS, and how BSS (including natural semantics) can be encoded as CRS. The connections permit the use of proper variables and substitution in both formalisms.

## 1 Introduction

A primary concern of Peter Mosses has always been to make it possible to use formal semantic as the base for implementations of programming languages. This has led to the seminal experiments in using mathematical notation [9] as well as more intuitive and modular operational notation [10].

The work presented here grew out of discussions with Peter (as part of the first author's M.Sc. thesis work) on whether it would be useful to permit *mixing* different traditions of formal notations in specifications? One obstacle is, of course, that then the correspondences between the involved formalisms must be fully understood. In this work we discuss the precise relationship between two such traditions: *rewriting* and *big step semantics*.

Rewriting uses *rewrite rules* to describe each possible computation step; the result of a computation is found by repeatedly rewriting until the “normal form”, that cannot be further rewritten, is reached. The rewriting formalism is good when the focus is on equational reasoning.

Term rewrite systems (TRS) are systems where rewrite rules are applied to tree-like structures, permitting rewriting of *any* fragment of the term. Combinatory reductions systems (CRS) might be seen as an extension of TRS with the notion of variable binding as known from  $\lambda$ -calculus. As well as having function symbols, binding variables and a notion of abstraction, CRS have meta-variables. Definitions and terminology are given in Section 2, where the fact that TRS are a special kind of CRS is also explained. CRS are excellent for abstract

---

\* Parts of this work done while an M.Sc. student with Peter Mosses at DAIMI, University of Aarhus.

descriptions of simplification problems. For instance the  $\lambda$ -calculus can be described by a CRS with just a single rule:  $(\lambda x.Z(x))(Y) \rightarrow Z(Y)$ .

In contrast, big step semantics (BSS) uses logic *axioms* and *inference rules* to describe complete computations in the sense that the proof trees that can be constructed with the rules have as their final conclusion some complete reduction. In addition, it is traditional for BSS to pay attention to the *structural operational* aspects of the specifications, following the original advice of Plotkin to make the choice of which rule needs to be inserted in the proof tree depend only on the syntax of the investigated program fragment [12].<sup>1</sup> Kahn [4] evolved this to *natural semantics*, where proof trees explicitly correspond to complete computations rather than computation steps, and where it is understood that the proof tree construction process is operational, *i.e.*, can be performed without backtracking. We shall retain the operationality requirement here, however, not the distinction between expressions and values since it is our aim to fully model the rewriting relation.

The requirement that entire computations are described means that if a BSS is to be, say, transitive, then that BSS must either contain a rule for transitivity or it must be possible to combine two evaluations in the appropriate way to obtain transitivity. The main benefit of using BSS over CRS is that in BSS evaluation starts at the outermost symbol of the term rather than at any subterm. This means that given a term with some root symbol, one need only consider inference rules with conclusions related to this symbol, thus we can consider the set of inference rules as a kind of decision table. This lessens the search space when looking for an inference rule to apply, even though the BSS have more rules.

Thus we are interested in deriving methods for constructing big step semantics corresponding to some CRS, and vice versa. In this paper such methods are described and proved to be correct with respect to specific correspondences between big step semantics and CRS.

A simple rewrite system might operate on addition expressions with syntax

$$e ::= e + e \mid n$$

(with  $n$  integer literals) and a infinite collection of rewrite rules described by

$$n_1 + n_2 \rightarrow n$$

with one rule for each of all the possible combinations of the integers such that  $n = n_1 + n_2$ . An inference system for computing all possible normal forms might look like this:

$$\frac{e_1 \Rightarrow n_1 \quad e_2 \Rightarrow n_2}{e_1 + e_2 \Rightarrow n} \quad \text{if } n = n_1 + n_2$$

$$\overline{n \Rightarrow n}$$

---

<sup>1</sup> When Plotkin wrote the report “A Structural Approach to Operational Semantics” [12], he described which was later to be known as *structural operational semantics* or *small step semantics*. Using the Stack-Memory-Code machine (SMC) as an example, Plotkin described how individual computation steps could be described in a syntax directed manner where the semantics of some composite statement is given by the semantics of its components.

(again written as a schema corresponding to an infinity of inference rules and axioms). The inference rules allow construction of proofs, which captures all the possible reductions to normal form. If we also wish to capture reductions to non-normal forms, we should include the rules

$$\frac{e_1 \Rightarrow n_1}{e_1 + e_2 \Rightarrow n_1 + e_2}$$

$$\frac{e_2 \Rightarrow n_2}{e_1 + e_2 \Rightarrow e_1 + n_2}$$

*Remark 1 (on rewriting logic).* In [8] Meseguer defines the notion of *rewriting logic* by giving some rules of deduction. This set of rules consists of rules for *reflexivity*, *congruence*, *transitivity* and *replacement*. With these rules the meaning of rewriting has been made explicit as  $\rightarrow$  is the transitive, reflexive closure of  $\rightarrow$ , which allows single steps of rewriting in any context. Meseguer uses rewriting logic to describe concurrent rewriting. This does give him a different perspective on things, but it is still worth comparing the two systems.

The BSS that we will construct here have no rule for transitivity and the general rules for reflexivity and congruence are only used when proving an evaluation to a non-normal form.

On the other hand Meseguer's system does need transitivity, even for proving evaluation to some normal form. Without transitivity, no more than one reduction could be made of any one subterm, thereby not capturing a case like  $f \rightarrow g \rightarrow h$ .

While we intended to build big step semantics focusing on the connections between the reduction sequence and the proofs, Meseguer's rewriting logic is constructed with a different purpose. Where we wished to be able to distinguish proofs of evaluations to normal forms, thereby having to avoid rules like those of transitivity and congruence for all function symbols, Meseguer wanted to be able to express concurrent rewriting in a simple, yet thorough form. The differences between those two intentions are visible in the definitions of the systems.

*Overview.* The rest of this paper is structured as follows: In Section 2 we briefly describe the notion of combinatory reduction systems (CRS) and present an example, the  $\lambda$ -calculus. Section 3 presents the method for deriving a BSS corresponding to some CRS, defines the equivalence and proves that the method produces a BSS with the desired properties. Section 4 presents a method for deriving a CRS from a BSS and discusses an example of this method. Finally Section 5 concludes and presents ideas for further work.

## 2 Combinatory Reduction Systems

First a short summary of the definition of CRS, based on [14], [5] and [6].

**Definition 1 (preterms over  $\Sigma$ ).** *Given some signature  $\Sigma$  consisting of a set of variables  $x$ , function symbols  $F^n$ , and meta-variables  $Z^n$ , the set  $P_\Sigma$  of preterms over  $\Sigma$  is defined by*

$$t ::= x \mid F^n(t_1, \dots, t_n) \mid Z^n(t_1, \dots, t_n) \mid x.t$$

In practice we shall use the syntactic constructs of a particular language as the function symbols (such as  $\lambda$ ) instead of  $F^n$ , and we shall use the nonterminal names (such as  $e$ ) instead of  $Z^n$ . We omit the arity of function symbols and meta-variables and parentheses whenever these are clear from the context, and abbreviate sequences of things as “vectors” and write, for example  $F^n(t)$  for  $F^n(t_1, \dots, t_n)$  and  $\bar{x}.t$  for  $x_1.x_2 \dots x_n.t$ .

**Definition 2 (closed preterms).** A preterm is said to be closed if it contains no free variables, where the set of free variables is given by  $\text{fv}(x) = \{x\}$ ,  $\text{fv}(F(t_1, \dots, t_n)) = \bigcup_{i=1}^n \text{fv}(t_i)$ ,  $\text{fv}(Z(t_1, \dots, t_n)) = \bigcup_{i=1}^n \text{fv}(t_i)$ , and  $\text{fv}(x.t) = \text{fv}(t) \setminus \{x\}$ .

**Definition 3 (meta-terms and terms).** Meta-terms, denoted  $M_\Sigma$ , are simply closed preterms over  $\Sigma$  and terms, denoted  $T_\Sigma$ , are meta-terms with no meta-variables.

The above definitions all relate to unsorted signatures, although by requiring all terms to be well formed they extend to many-sorted signatures. This is in fact the case throughout the paper.

*Example 1 ( $\lambda$  calculus).* The signature for the ordinary  $\lambda\beta$ -calculus is described by:

$$\begin{aligned}\Sigma_1 &= \{\lambda\} \\ \Sigma_2 &= \{\text{app}\}\end{aligned}$$

To conform better to traditional notation, we write  $\text{app}(X, Y)$  as  $XY$  and  $\lambda(x.X)$  as  $\lambda x.X$ .

The following illustrates preterms, meta-terms and terms respectively:

- $(\lambda(x.Z(x)))(Y(y))$  is a preterm (it contains the free variable  $y$ ).
- $y.(\lambda(x.Z(x))(Y))$  is a meta-term (has no free variables but does contain a meta-application).
- $(\lambda(x.F(x)))(\lambda(y.G(y)))$  is a term (neither free variables nor meta-applications).

**Definition 4 (rule).** A rule takes the form  $p \rightarrow c$  with  $p, c \in M_\Sigma$ , where  $p$  is a pattern, that is has a function symbol at the root and all arguments of meta-applications are distinct variables, and  $c$  a contractum of  $p$ , that is only contains meta-variables occurring in  $p$ .

The above definition of CRS is un-sorted, but to obtain a sorted CRS we simply restrict the terms to be well-sorted according to a many-sorted signature. Thus the sorted CRS is simply a subsystem of the un-sorted CRS (provided it is proved that each rule is sort-preserving).

*Example 2 ( $\lambda$  calculus).* The  $\lambda$  calculus can be described by the signature of Example 1 and the reduction rule

$$(\lambda x.Z(x))Y \rightarrow Z(Y)$$

**Definition 5 (valuation).** A valuation,  $\sigma_\Sigma$ , is a map from meta-variables to terms, such that  $\sigma_\Sigma(Z^n) = t(x_1, \dots, x_n)$ , where  $t$  is a preterm and  $x_1, \dots, x_n$  are distinct variables. The valuation  $\sigma_\Sigma$  extends to a map from meta-terms to terms:

1.  $\forall x \in \text{Var} : \sigma_\Sigma(x) = x$
2.  $\forall F^n \in \Sigma, \forall t_1, \dots, t_n \in M_\Sigma : \sigma_\Sigma(F(t_1, \dots, t_n)) = F(\sigma_\Sigma(t_1), \dots, \sigma_\Sigma(t_n))$
3.  $\forall Z^\Sigma \in \Sigma, \forall t_1, \dots, t_n \in M_\Sigma : \sigma_\Sigma(Z(t_1, \dots, t_n)) = \sigma_\Sigma(Z)(\sigma_\Sigma(t_1), \dots, \sigma_\Sigma(t_n))$
4.  $\forall x \in \Sigma, \forall t \in M_\Sigma : \sigma_\Sigma(x.t) = x.\sigma_\Sigma(t)$

Thus  $\sigma_\Sigma$  is determined by its value on the meta-variables.

In order to define rewrites we need a way to split a term into a context and contained fragments. We write  $t = C[t_1, \dots, t_n]$  when  $t$  can be separated into the context  $C[]$  and the fragments  $t_1, \dots, t_n$ .

**Definition 6 (rewrite step).** A reduction rule  $p \rightarrow c$  defines a reduction relation that includes all “rewrite steps” of the form

$$C[\sigma(p)] \rightarrow C[\sigma(c)]$$

where  $C[]$  is a context and  $\sigma$  a valuation that maps all meta-variables in  $p$ .

As usual  $\rightarrow \cdot \rightarrow$  is composition,  $\rightarrow^n$  denotes  $n$  single steps, and  $\rightarrow\!\!\!$  is the reflexive and transitive closure of  $\rightarrow$ . In addition, a term  $t \in T$  is in *normal form* if there is no  $s \in T$  such that  $t \rightarrow s$ , and if  $s \rightarrow\!\!\! t$  is and  $t$  is a normal form we write  $s \rightarrow\!\!\! t$ . Now we can put the pieces together.

**Definition 7 (combinatory reduction system).** A combinatory reduction system (CRS) is a pair  $\langle T, \rightarrow \rangle$  where  $T$  is a set of terms and  $\rightarrow$  is a reduction relation defined as the union of the rewrite steps allowed by a collection of CRS rules.

Finally, we need various constraints.

**Definition 8 (orthogonal).** Let there be given a CRS  $\langle M_\Sigma, \rightarrow \rangle$  with two reduction rules  $p_1 \rightarrow c_1$  and  $p_2 \rightarrow c_2$ . If  $p_2$  matches some non-variable subterm of  $p_1$  then there exists a context  $C[]$ , a term  $p$  and a valuation  $\sigma_\Sigma$  such that  $p_1 = C[p]$  and  $\sigma_\Sigma(p) = \sigma_\Sigma(p_2)$ . Now  $\sigma_\Sigma(C[p])$  can be reduced in either of two ways:  $\sigma_\Sigma(C[p]) \rightarrow \sigma_\Sigma(c_1)$  or  $\sigma_\Sigma(C[p]) \rightarrow \sigma_\Sigma(c_2)$ . The pair of reducts  $\langle \sigma_\Sigma(c_1), \sigma_\Sigma(C[c_2]) \rangle$  is called a critical pair. Whenever the two rewrite rules coincide,  $p_2$  must be a proper subterm of  $p_1$ , i.e.,  $p_1 \neq p_2$ . A CRS is orthogonal if it has no critical pairs.

**Remark 2 (term rewrite system).** First order term rewrite systems as described in [6] are CRS restricted by disallowing meta-abstraction, therefore TRS have no binding variables and their meta-variables are of arity 0. These are often referred to simply as variables. Specifically, the preterms for TRS are given by

$$t ::= F^n(t_1, \dots, t_n) \mid Z$$

### 3 From CRS to BSS

In this section we construct inference rules for a big step semantic description that is equivalent to the rewrite rules for a given combinatory reduction system. The idea is the following: given a rewrite relation  $\rightarrow$ , we construct a proof system such that the judgment

$$\phi \vdash s \Rightarrow t$$

is provable when  $s \rightarrow t$  and  $\phi$  contains all the free variables of  $s$  and  $t$ .

The notion of BSS exploited by the construction is quite standard, except we permit direct use of substitution in inference rule premises. (The full details can be found in the first author's M.Sc. thesis [3].) Last, we shall see how the BSS and CRS may be said to be equivalent.

The constructed BSS has three different kinds of rules. First for each reduction rule in the CRS there is a corresponding inference rule, which is used in proofs once for each rewrite. Then for each term not matching any reduction rule, there is an inference rule to permit reduction inside the irreducible form, and finally we permit reflexivity and reductions in any construction. If the alphabet of function symbols is finite, this can be described finitely using variables.

**Definition 9 (big step semantics construction).** *Given a CRS  $C = \langle T_\Sigma, R \rangle$  we describe how to construct a BSS related to  $C$ ,  $BSS(C)$ .*

**Type 1.** Consider each of the CRS rewrite rules to be of the form

$$F^n(C_1[Z_{11}[\bar{x}_{11}], \dots], \dots, C_n[Z_{n1}[\bar{x}_{n1}], \dots]) \rightarrow c$$

where the  $Z_{ij}[\bar{x}_{ij}]$  fragments capture all the pattern meta-applications. Then add the following corresponding inference rule to the BSS:

$$\frac{\phi \vdash U_1 \Rightarrow C_1[Z_{11}[\bar{x}_{11}], \dots] \quad \dots \quad \phi \vdash U_n \Rightarrow C_n[Z_{n1}[\bar{x}_{n1}], \dots] \quad \phi \vdash c \Rightarrow V}{\phi \vdash F^n(\bar{U}) \Rightarrow V}$$

except in case a  $C_i[Z_{i1}[\bar{x}_{i1}], \dots]$  is simply  $U_i$  then omit the  $\phi \vdash U_i \Rightarrow C_i[Z_{i1}[\bar{x}_{i1}], \dots]$  premise from the rule.

**Type 2.** For each term  $t$  which does not match any reduction rule: If  $t$  is of the form  $F(t_1, \dots, t_n)$  the following rule must be in the BSS:

$$\frac{\phi \vdash U_1 \Rightarrow t_1 \quad \dots \quad \phi \vdash U_n \Rightarrow t_n}{\phi \vdash F(\bar{U}) \Rightarrow F(\bar{t})}$$

If  $t$  is of the form  $x$  the following rule must be in the big step semantics:

$$\phi \vdash x \Rightarrow x \quad \text{if } x \in \phi$$

If  $t$  is of the form  $x.t'$  the following rule must be in the big step semantics:

$$\frac{x \cdot \phi \vdash t' \Rightarrow X}{\phi \vdash x.t' \Rightarrow x.X}$$

**Type 3.** Standard rules, which must always be present:

$$\phi \vdash U \Rightarrow U$$

For each  $F^n$  ( $n \geq 1$ ) the following rule must be in the big step semantics:

$$\frac{\phi \vdash t_1 \Rightarrow t'_1 \quad \dots \quad \phi \vdash t_n \Rightarrow t'_n}{\phi \vdash F(t_1, \dots, t_n) \Rightarrow F(t'_1, \dots, t'_n)} \text{ if } \exists i \in \{1, \dots, n\} : t_i \neq t'_i$$

We see that it is mechanical to construct such a BSS given a CRS. The restriction on the premise of Type 1 rules ensures that we can avoid unnecessary evaluations which may otherwise cause non-termination.

*Example 3* ( $\lambda$ -calculus and BSS). For the  $\lambda$ -calculus (Example 2), the above construction gives the following BSS:

**Type 1:**

$$\frac{\phi \vdash U_1 \Rightarrow \lambda x. Z(x) \quad \phi \vdash Z(U_2) \Rightarrow X}{\phi \vdash U_1 U_2 \Rightarrow X} \text{ 1a}$$

**Type 2:**

$$\begin{aligned} & \frac{\phi \vdash U \Rightarrow X}{\phi \vdash \lambda U \Rightarrow \lambda X} \text{ 2a} \\ & \frac{\phi \vdash U_1 \Rightarrow x \quad \phi \vdash U_2 \Rightarrow Z}{\phi \vdash U_1 U_2 \Rightarrow xZ} \text{ 2b} \\ & \frac{\phi \vdash U_1 \Rightarrow Z_1 Z_2 \quad \phi \vdash U_2 \Rightarrow X}{\phi \vdash U_1 U_2 \Rightarrow (Z_1 Z_2)X} \text{ 2c} \\ & \frac{x \cdot \phi \vdash U \Rightarrow X}{\phi \vdash x.U \Rightarrow x.X} \text{ 2d} \end{aligned}$$

**Type 3:**

$$\begin{aligned} & \phi \vdash U \Rightarrow U \text{ 3a} \\ & \frac{\phi \vdash U \Rightarrow X}{\phi \vdash \lambda U \Rightarrow \lambda X} \text{ 3b} \\ & \frac{\phi \vdash U_1 \Rightarrow X_1 \quad \phi \vdash U_2 \Rightarrow X_2}{\phi \vdash U_1 U_2 \Rightarrow X_1 X_2} \text{ 3c} \end{aligned}$$

The construction of the BSS from a CRS can easily be automated provided a method for higher order unification is already present. Only the construction of type 2 rules present a challenge as we must calculate the set of terms not matching any of the reduction rules. For each reduction rule we can list the terms (with use of variables) which do not match the left hand side. In the  $\lambda$ -calculus there is just one reduction rule with the left hand side  $(\lambda x. Z(x))(Y)$ . When there is more than one reduction rule, we use unification to find terms present in all of these sets. Those terms then give rise to the type 2 rules.

Now let us turn to the equivalence of a CRS and the constructed BSS. First we will define what we mean by *equivalence*, then sketch the proof that this equivalence does indeed hold for the construction presented above.

**Definition 10 (equivalence).** A CRS  $C = \langle T_\Sigma, \rightarrow \rangle$  and a BSS  $B$  are equivalent whenever  $\forall s, t \in T_\Sigma$  the following holds:

$$s \twoheadrightarrow t \text{ in } C \text{ if and only if } s \Rightarrow t \text{ in } B.$$

The definition of equivalence says that the two systems are equivalent if and only if any reduction corresponds to a proof in the big step semantics. However it does not say anything about uniqueness of either the reduction sequence or the proof. In fact several different reduction sequences may give rise to the same proof.

**Theorem 1 (equivalence between a CRS,  $C$ , and  $BSS(C)$ ).** Let  $C$  be a CRS. Then  $C$  and  $BSS(C)$  are equivalent.

The proof of this theorem is constructive, i.e., given some reduction sequence  $s \twoheadrightarrow t$ , one can construct a proof of  $s \Rightarrow t$  by following the proof and vice versa.

For simplicity, we will now assume notation as above and will no longer refer to the sets of terms over some signature. This when considering a reduction  $s \rightarrow t$  and a proof  $\phi \vdash s \Rightarrow t$  we will assume that the reduction comes from  $C = \langle T_\Sigma, \rightarrow \rangle$  and the proof from  $BSS(C)$ .

### 3.1 Example

Before we go on to give an outline of the proof of Theorem 1, let us illustrate the construction of a proof from a reduction sequence by an example. Of course, given Theorem 1 one is free to use the constructed BSS without references to particular reductions sequences. This example shows the principles of the proof of Theorem 1.

*Example 4 (sample reduction).* Referring to the  $\lambda$ -calculus of Example 3, consider the following reduction:

$$\lambda x.(\lambda y.(\lambda z.zz)y)x \rightarrow \lambda x.(\lambda y.yy)x \rightarrow \lambda x.xx$$

We treat the reduction sequence from the end, so first we want to construct a proof  $P_1$  of  $\lambda x.xx \Rightarrow \lambda x.xx$ . Then we construct a proof  $P_{2a}$  of  $x \vdash (\lambda y.yy)x \Rightarrow xx$  as this is the redex of the last reduction step. By combining these first two proofs we obtain a proof  $P_{2b}$  of  $\lambda x.(\lambda y.yy)x \Rightarrow \lambda x.xx$ . In the same way, we construct a proof  $P_{3b}$  of  $\lambda x.(\lambda y.(\lambda z.zz)y)x \Rightarrow \lambda x.(\lambda y.yy)x$  and finally we combine  $P_{2b}$  and  $P_{3b}$  to obtain the proof of  $\lambda x.(\lambda y.(\lambda z.zz)y)x \Rightarrow \lambda x.xx$ .

As  $\lambda x.xx$  is in normal form,  $P_1$  uses only type 2 rules:

$$\frac{\overline{x \vdash x \Rightarrow x} \quad T2 \quad \overline{x \vdash x \Rightarrow x} \quad T2}{\frac{\overline{x \vdash xx \Rightarrow xx} \quad T2}{\frac{\vdash x.xx \Rightarrow x.xx \quad T2}{\vdash \lambda x.xx \Rightarrow \lambda x.xx \quad T2}}} \quad (P_1)$$

Next, consider the redex  $(\lambda y.y)y$ . We use exactly the instance of the type 1 rules which corresponds to the reduction  $(\lambda y.y)y \rightarrow yy$ :

$$\frac{\begin{array}{c} y \cdot x \vdash y \Rightarrow y \text{ T2} \\ y \cdot x \vdash y \Rightarrow y \text{ T2} \end{array}}{\frac{\begin{array}{c} y \cdot x \vdash yy \Rightarrow yy \text{ T2} \\ x \vdash y.yy \Rightarrow y.yy \text{ T2} \end{array}}{\frac{x \vdash \lambda y.yy \Rightarrow \lambda y.yy \text{ T2}}{\frac{\begin{array}{c} x \vdash x \Rightarrow x \text{ T2} \\ x \vdash xx \Rightarrow xx \text{ T2} \end{array}}{x \vdash (\lambda y.yy)x \Rightarrow xx \text{ T1}}}}}} \quad (P_{2a})$$

Now combine  $P_1$  and  $P_{2a}$ :

$$\frac{\begin{array}{c} y \cdot x \vdash y \Rightarrow y \text{ T2} \\ y \cdot x \vdash y \Rightarrow y \text{ T2} \end{array}}{\frac{\begin{array}{c} y \cdot x \vdash yy \Rightarrow yy \text{ T2} \\ x \vdash y.yy \Rightarrow y.yy \text{ T2} \end{array}}{\frac{x \vdash \lambda y.yy \Rightarrow \lambda y.yy \text{ T2}}{\frac{\begin{array}{c} x \vdash x \Rightarrow x \text{ T2} \\ x \vdash xx \Rightarrow xx \text{ T2} \end{array}}{\frac{x \vdash (\lambda y.yy)x \Rightarrow xx \text{ T1}}{\frac{\vdash x.(\lambda y.yy)x \Rightarrow x.xx \text{ T2}}{\vdash \lambda x.(\lambda y.yy)x \Rightarrow \lambda x.xx \text{ T2}}}}}}}} \quad (P_{2b})$$

The last redex is  $(\lambda z.zz)y$  which reduces to  $yy$ . Again the corresponding instance of the type 1 rules is used:

$$\frac{\begin{array}{c} z \cdot y \cdot x \vdash z \Rightarrow z \text{ T2} \\ z \cdot y \cdot x \vdash z \Rightarrow z \text{ T2} \end{array}}{\frac{\begin{array}{c} z \cdot y \cdot x \vdash zz \Rightarrow zz \text{ T2} \\ y \cdot x \vdash z.zz \Rightarrow z.zz \text{ T2} \end{array}}{\frac{y \cdot x \vdash \lambda z.zz \Rightarrow \lambda z.zz \text{ T2}}{\frac{\vdash x.(\lambda z.zz)y \Rightarrow yy \text{ T1}}{\vdash \lambda x.(\lambda z.zz)y \Rightarrow \lambda x.xx \text{ T2}}}}}} \quad (P_{3a})$$

As  $\lambda x.(\lambda y.yy)x$  is not in normal form, we use type 3 rules to build the context of  $P_{3a}$ :

$$\frac{\begin{array}{c} P_{3a} \\ \vdash y.(\lambda z.zz)y \Rightarrow y.yy \text{ T3} \end{array}}{\frac{\vdash \lambda y.(\lambda z.zz)y \Rightarrow \lambda y.yy \text{ T3}}{\frac{\vdash x.(\lambda y.(\lambda z.zz)y)x \Rightarrow (\lambda y.yy)x \text{ T3}}{\frac{\vdash x.(\lambda y.(\lambda z.zz)y)x \Rightarrow x.(\lambda y.yy)x \text{ T3}}{\vdash \lambda x.(\lambda y.(\lambda z.zz)y)x \Rightarrow \lambda x.(\lambda y.yy)x \text{ T3}}}}}} \quad (P_{3b})$$

Finally, combine  $P_{2b}$  and  $P_{3b}$  to obtain a proof which has exactly the two type 1 rules corresponding to the two reductions:

$$\begin{array}{c}
 \frac{P_{3a}}{x \vdash y.(\lambda z.zz)y \Rightarrow y.yy} T2 \quad \frac{x \vdash x \Rightarrow x \quad x \vdash x \Rightarrow x}{x \vdash xx \Rightarrow xx} T2 \\
 \frac{}{\vdash x.(\lambda y.(\lambda z.zz)y)x \Rightarrow x.xx} T2 \quad \frac{x \vdash (\lambda y.(\lambda z.zz)y)x \Rightarrow xx}{\vdash x.(\lambda y.(\lambda z.zz)y)x \Rightarrow \lambda x.xx} T1 \\
 \frac{}{\vdash x.(\lambda y.(\lambda z.zz)y)x \Rightarrow \lambda x.xx} T2
 \end{array}$$

### 3.2 Outline of Proof of Equivalence

We present just an outline of the proof of equivalence between a CRS and the corresponding BSS. The full proof can be found in [3]. There are two parts to the proof: one is to prove that for any proof  $\phi \vdash s \Rightarrow t$  in the BSS there is a reduction  $s \rightarrow t$  in the CRS, the other is to prove that for any reduction  $s \rightarrow t$  there is a proof  $\phi \vdash s \Rightarrow t$  in the BSS.

All the propositions and lemmas used in the proof of Theorem 1 are stated and a brief description of each proof given.

The first proposition states that  $\Rightarrow$  is transitive, that is if proofs of  $\phi \vdash s_1 \Rightarrow s_2$  and  $\phi \vdash s_2 \Rightarrow s_3$  exists then so does a proof of  $\phi \vdash s_1 \Rightarrow s_3$ . This is not trivial since the BSS does not explicitly contain transitivity rules.

The first lemma says that any proof  $\phi \vdash s \Rightarrow t$  corresponds to some reduction  $s \rightarrow t$ . Intuitively, this is what we would expect as each use of a type 1 rule corresponds to a reduction step whereas both type 2 and 3 rules are used for building the context in which reduction takes place. This concludes the first half of the proof of Theorem 1.

The second proposition says that for any term  $t$  in normal form a proof of  $\vdash t \Rightarrow t$  in the BSS can be constructed using only type 2 rules.

Finally, the second lemma says that corresponding to any reduction  $s \rightarrow t$  there is a proof of  $\vdash s \Rightarrow t$  in the BSS. This then completes the proof of Theorem 1.

**Proposition 1 (transitivity).** *The BSS consisting of the rules described in Definition 9 is transitive, that is  $\forall r, s, t \in T$  if  $\phi \vdash r \Rightarrow s$  and  $\phi \vdash s \Rightarrow t$  then  $\phi \vdash r \Rightarrow t$ .*

*Proof.* The proof is by induction on the shape of the proof of  $r \Rightarrow s$ .

**Base case:** The base cases are the axioms. The only axioms are type 2 rules for constant function symbols and variables and the type 3 rule for reflexivity.

In each of these cases, we see that  $r = s$ , and as we already know that  $s \Rightarrow t$ , it follows that  $r \Rightarrow t$ .

**Hypothesis:** Assume that for the premises of the rules, the proposition is true.

**Inductions Step:** Four different kinds of rules might be at the bottom of the proof tree: type 1 rules, type 2 rules for non-constant function symbols or for abstraction or type 3 rules for contexts. Consider each of these four separately:

**Type 1:** Follows immediately from the induction hypothesis.

**Type 2 for non-constant function symbols:** Follows from induction on the shape of the proof of  $s \Rightarrow t$ :

**Base case of the sub proof:** The only axiom is the type 3 rule for reflexivity. Then  $s = t$  and thus the proof of  $r \Rightarrow t$  is identical to that of  $r \Rightarrow s$ .

**Hypothesis of the sub proof:** Assume that for all premises  $X_i \Rightarrow Y_i$  of the proof of  $s \Rightarrow t$ , the statement holds.

**Induction step for the sub proof:** Type 1, type 2 for non-constant function symbols or type 3 for context rules might be used here. In any of these cases the proof follows directly from the two induction hypotheses.

**Type 2 for abstraction:** That is  $r = x.C_1$ ,  $s = x.C_2$  and  $t = x.C_3$  as no rewrite rule can have abstraction at the root. The proof of  $s \Rightarrow t$  must have the type 2 rule for abstraction or the type 3 rule for reflexivity at the bottom. In either case, the proof follows immediately from the induction hypothesis.

**Type 3 for contexts:** The proof of  $s \Rightarrow t$  must have either of the following rules at the bottom: type 1, type 2 for non-constant function symbols, type 3 for contexts or type 3 for reflexivity. The proof follows from induction on the shape of the proof of  $s \Rightarrow t$ :

**Base case of the sub proof:** The only axioms are by the type 3 rule for reflexivity. If this is used then  $s = t$  and the proof of  $r \Rightarrow s$  is also a proof of  $r \Rightarrow t$ .

**Hypothesis of the sub proof:** Assume that for all premises  $X_i \Rightarrow Y_i$  of the proof of  $s \Rightarrow t$ , the statement holds.

**Induction step of the sub proof:** Either of type 1, type 2 for non-constant function symbol or type 3 for context rules might be used here. In any of these cases the proof follows immediately from the two induction hypotheses.

Thus the BSS is transitive. □

**Lemma 1.** *For any  $s, t \in T$ , given a proof of  $\phi \vdash t \Rightarrow s$  there is a reduction  $t \rightarrow s$ .*

*Proof.* To prove this, simply use induction on the shape of the proof of  $\phi \vdash t \Rightarrow s$ :

**Base case:** The base cases are the axioms. The only axioms are type 2 rules for constant function symbols and for variables and the type 3 rule for reflexivity. All these rules are of the form  $t \Rightarrow t$  and a corresponding reduction is thus  $t \rightarrow^0 t$ .

**Hypothesis:** Assume that given proofs of  $\phi \vdash U_i \Rightarrow C_i [\bar{Y}]$  for  $i \in \{1, \dots, n\}$ , of  $x \cdot \phi \vdash U(x) \Rightarrow V(x)$ , and of  $\phi \vdash C' [\bar{Y}] \Rightarrow s$  we have corresponding reductions.

**Induction step:** Given a proof of  $\phi \vdash t \Rightarrow s$ , we have to produce a reduction  $t \rightarrow s$ . Either of type 1 or type 2 rules, for non-constant function symbols or abstraction, or type 3 rules, for contexts, might be at the bottom of the

proof. In any of these cases, the existence of a reduction follows immediately from the induction hypothesis.  $\square$

**Proposition 2.** *For any  $t \in T$  in normal form we can construct a proof of  $\phi \vdash t \Rightarrow t$  by using only type 2 rules.*

*Proof.* To prove this we use induction on the depth of the term  $t$ .

**Base case:** Whether  $t$  is a constant function symbol or a variable  $\phi \vdash t \Rightarrow t$  is proved by a single type 2 rule.

**Hypothesis:** Assume we have constructed proofs,  $P_1, \dots, P_n$  of  $\vdash t_1 \Rightarrow t_1 \dots \vdash t_n \Rightarrow t_n$  and  $P_{t'}$  of  $\vdash t' \Rightarrow t'$ .

**Induction step:** We know that each  $t$  in normal form matches some type 2 rule. In any of these cases the proof follows from the induction hypothesis as  $t$  is in normal form only if each subterm of  $t$  is in normal form.  $\square$

**Lemma 2.** *For any  $s, t \in T$ , given a reduction  $t \rightarrow s$  there is a proof of  $t \Rightarrow s$ .*

*Proof.* To show that given a reduction  $t \rightarrow^n s$  we can construct a proof of  $\vdash t \Rightarrow s$ , we use induction on the length of the reduction sequence.

**Base cases:**  $n = 0$

Then  $t \rightarrow^0 s$ , so  $t = s$ . If  $t$  is in normal form use Proposition 2 to construct a proof of  $\vdash t \Rightarrow t$ . If  $t$  is not in normal form, use the type 3 rule for reflexivity to prove  $\vdash t \Rightarrow t$ .

$n = 1$

Then  $C_1[s] \rightarrow C_1[s']$  where  $s$  is the redex being rewritten. A proof of  $\vdash C_1[s'] \Rightarrow C_1[s']$  can be constructed and so can a proof of  $\vdash s \Rightarrow s'$  as there is a type 1 rule corresponding to the rewrite rule that  $s$  matches. If  $C_1[s']$  is in normal form a proof of  $\vdash s' \Rightarrow s'$  is part of the proof of  $\vdash C_1[s'] \Rightarrow C_1[s']$ . Now insert the proof of  $\vdash s \Rightarrow s'$  instead of the proof of  $\vdash s' \Rightarrow s'$  to obtain a proof of  $\vdash C_1[s] \Rightarrow C_1[s']$ . If  $C_1[s']$  is not in normal form, the proof of  $\vdash C_1[s'] \Rightarrow C_1[s']$  is by the type 3 rule for reflexivity. Use the type 3 rule for context repeatedly to obtain a proof

$$\frac{\vdash a_1 \Rightarrow a'_1 \dots \vdash a_n \Rightarrow a'_n}{\vdash F(a_1, \dots, a_n) \Rightarrow F(a'_1, \dots, a'_n)}$$

**Hypothesis:** Assume for  $n \geq 1$  that given  $t \rightarrow^n s$  we can construct a proof  $\vdash t \Rightarrow s$ .

**Induction step:** Consider a reduction of length  $n+1$ ,  $C_1[s] \rightarrow^n C_2[t'] \rightarrow C_2[t]$  where  $s$  is the first redex to be rewritten. By the induction hypothesis proofs of  $\vdash C_1[s] \Rightarrow C_2[t']$  and of  $\vdash C_2[t'] \Rightarrow C_2[t]$  exists. As the BSS is transitive by Proposition 1 a proof of  $\vdash C_1[s] \Rightarrow C_2[t]$  now exists.  $\square$

The proof of Theorem 1 can now be completed:

*Proof.* Let  $t, s \in T$ . Then Lemma 1 says that given a proof of  $\vdash t \Rightarrow s$  there is a reduction  $t \rightarrow s$  and Lemma 2 says that given a reduction  $t \rightarrow s$  there is a proof of  $\vdash t \Rightarrow s$ . Thus  $t \rightarrow s$  if and only if  $\vdash t \Rightarrow s$ .  $\square$

As each of the lemmas have constructive proofs, we now have methods for constructing a proof corresponding to a given reduction and vice versa.

## 4 From BSS to CRS

In this section we study how a BSS can be converted to a CRS by encoding the proof construction into rules with special function symbols. This only works for certain restricted classes of BSS, however. The mechanism generalizes existing methods for first order systems [16]. The method permits more general big step semantic specifications than generated in the previous section, for example specifications using a proper *environment* and separate source and value sorts, such as in natural semantics [4].

**Definition 11 (big step combinatory reduction system construction).** *Start with a BSS with inference rules and axioms for judgments. Without loss of generality we can assume the axioms have the form  $t_1 \Rightarrow t_2$  with  $t_1, t_2$  CRS meta-terms over signature  $\Sigma$ . Now construct the rules for the new big step semantics combinatory reduction system (BSS-CRS) rewrite relation as follows.*

1. *The terms of the generated CRS are over  $\Sigma_{!?} = \Sigma \cup \{ ?^1 \} \cup \{ !_n^{n+1} \mid n > 0 \}$ .*
2. *For each BSS axiom*

$$\overline{\bar{q}.Q \Rightarrow \bar{c}.C} \quad N$$

*include the CRS rewrite rule*

$$?( \bar{q}.Q ) \rightarrow \bar{c}.C \quad (N)$$

3. *For each BSS inference rule (with  $n \geq 1$  premises)*

$$\frac{\bar{q}_1.Q_1 \Rightarrow \bar{c}_1.C_1 \quad \dots \quad \bar{q}_i.Q_i \Rightarrow \bar{c}_i.C_i \quad \dots \quad \bar{q}_n.Q_n \Rightarrow \bar{c}_n.C_n}{\bar{q}.Q \Rightarrow \bar{c}.C} \quad N$$

*include the CRS rewrite rules*

$$?( \bar{q}.Q ) \rightarrow !_1( \bar{q}.Q, \bar{q}_1.?(Q_1) ) \quad (N_0)$$

$$!_1( \bar{q}.Q, \bar{c}_1.C_1 ) \rightarrow !_2( \bar{q}.Q, \bar{c}_1.C_1, \bar{q}_2.?(Q_2) ) \quad (N_1)$$

⋮

$$!_i( \bar{q}.Q, \bar{c}_1.C_1, \dots, \bar{c}_i.C_i ) \rightarrow !_i( \bar{q}.Q, \bar{c}_1.C_1, \dots, \bar{c}_i.C_i, \bar{q}_{i+1}.?(Q_{i+1}) ) \quad (N_i)$$

⋮

$$!_n( \bar{q}.Q, \bar{c}_1.C_1, \dots, \bar{c}_n.C_n ) \rightarrow \bar{c}.C \quad (N_n)$$

*This collection of rewrite rules as a CRS define the rewrite relations  $\rightarrow$  and  $\rightarrow\!\!\!\rightarrow$  as usual, in addition to  $\rightarrow\!\!\!\rightarrow|_{\Sigma}$  which is the restriction of the transitive closure to only terms with no  $?$  and  $!_i$  constructors.*

The translation effectively manifests all the intermediate stages of proof construction as terms, which means that the equivalence result we can prove is slightly weaker.

Also notice that the procedure may create duplicate rules in case there are overlapping inference rules, and so only works for inference systems where the premises can be instantiated strictly from left to right (“no backtracking”). We’ll make this precise below, after an example.

#### 4.1 Example

Consider the BSS created from the  $\lambda$ -calculus of Example 3. It has one axiom and 7 inference rules, which define the judgment  $\phi \vdash X \Rightarrow Y$  with  $\phi$  the list of binders for both sides: we will make this fit the required pattern by duplicating the binders, writing  $\bar{x} \vdash X \Rightarrow Y$  as  $\bar{x}.X \Rightarrow \bar{x}.Y$  (using the same binders on both sides of the reduction).

Following the above construction of a BSS-CRS from a BSS, we get the following reduction rules:

**Type 1:**

$$?(U_1 U_2) \rightarrow !_1(U_1 U_2, ?(U_1)) \quad (1a_0)$$

$$!_1(U_1 U_2, \lambda x. Z(x)) \rightarrow !_2(U_1 U_2, \lambda x. Z(x), ?(Z(U_2))) \quad (1a_1)$$

$$!_2(U_1 U_2, \lambda x. Z(x), X) \rightarrow X \quad (1a_2)$$

**Type 2:**

$$?( \lambda U ) \rightarrow !_1( \lambda U, ?(U) ) \quad (2a_0)$$

$$!_1( \lambda U, X ) \rightarrow \lambda X \quad (2a_1)$$

$$?(U_1 U_2) \rightarrow !_1(U_1 U_2, ?(U_1)) \quad (2b_0)$$

$$!_1(U_1 U_2, x) \rightarrow !_2(U_1 U_2, x, ?(U_2)) \quad (2b_1)$$

$$!_2(U_1 U_2, x, Z) \rightarrow xZ \quad (2b_2)$$

$$?(U_1 U_2) \rightarrow !_1(U_1 U_2, ?(U_1)) \quad (2c_0)$$

$$!_1(U_1 U_2, Z_1 Z_2) \rightarrow !_2(U_1 U_2, Z_1 Z_2, ?(U_2)) \quad (2c_1)$$

$$!_2(U_1 U_2, Z_1 Z_2, X) \rightarrow (Z_1 Z_2)X \quad (2c_2)$$

$$?(x.U(x)) \rightarrow !_1(x.U(x), x.? (U(x))) \quad (2d_0)$$

$$!_1(x.U(x), x.X(x)) \rightarrow x.X(x) \quad (2d_1)$$

**Type 3:**

$$?(U) \rightarrow U \quad (3a)$$

$$?( \lambda U ) \rightarrow !_1( \lambda U, ?(U) ) \quad (3b_0)$$

$$!_1( \lambda U, X ) \rightarrow \lambda X \quad (3b_1)$$

$$?(U_1 U_2) \rightarrow !_1(U_1 U_2, ?(U_1)) \quad (3c_0)$$

$$!_1(U_1 U_2, X_1) \rightarrow !_2(U_1 U_2, X_1, ?(U_2)) \quad (3c_1)$$

$$!_2(U_1 U_2, X_1, X_2) \rightarrow X_1 X_2 \quad (3c_2)$$

We see that the order of reduction with these rules corresponds exactly to always taking the left-most branch of the proof tree in the corresponding proof using the BSS rules of Example 3.

The first and obvious question is this: does this system define the same relation as the original  $\lambda$ -calculus CRS it was derived from? Specifically, is the transitive closure of the relation defined by the original CRS from Example 1, let us denote it  $\rightarrow_1$ , the same as the restriction of the above rewrite relation to terms without any of the special ? and ! constructors, denoted  $\rightarrow_2$ ? That this is the case can be seen from two facts:

1. Substitution – the key property of the original reduction rule – is encoded by the  $(1a_i)$  rules, which permits reducing  $?((\lambda x.Z(x))U_2)$  to  $Z(U_2)$ .
2. The collection of the remainder of the rules guarantees that the ? maker can move into any context.

Another use of this variation is that it provides a formal framework – rewriting – suitable for studying evaluation strategies in the form of the restriction of the system to subsets of the “reduction-in-context” rules [1] – a notoriously difficult problem – by making it manifest as the absence or presence of certain of the rewrite rules.

## 4.2 Equivalence

To prove equivalence we need to formalize the needed BSS restrictions.

**Definition 12 (complete big step semantics).** A BSS is complete if, when a term  $t$  matches the pattern part  $\bar{q}.Q$  of an axiom or inference rule then there exists a proof of  $t \Rightarrow t'$  for some  $t'$ .

**Definition 13 (context-free big step semantics).** Consider two rules

$$\frac{\bar{q}_1.Q_1 \Rightarrow \bar{c}_1.C_1 \cdots \bar{q}_n.Q_n \Rightarrow \bar{c}_n.C_n}{\bar{q}.Q \Rightarrow \bar{c}.C} \quad \frac{\bar{q}'_1.Q'_1 \Rightarrow \bar{c}'_1.C'_1 \cdots \bar{q}'_{n'}.Q'_{n'} \Rightarrow \bar{c}'_{n'}.C'_{n'}}{\bar{q}'.Q' \Rightarrow \bar{c}'.C'}$$

(or axioms, respectively, if  $n$  or  $n'$  are zero), ordered such that  $n \leq n'$ .

- If a term  $\bar{x}.t$  exists that instantiates both  $\bar{q}.Q$  and  $\bar{q}'.Q'$ , then the rules are said to overlap for  $\bar{x}.t$ .
- Furthermore, if the instantiation of the premises have  $\bar{q}_1.Q_1 = \bar{q}'.Q'$ ,  $\bar{c}_1.C_1 = \bar{c}'.C'$ , up to query of the last premise,  $\bar{q}_n.Q_n = \bar{q}'_{n'}.Q'_{n'}$ , then the overlap is trivial.
- If all overlaps of all pairs of rules in a BSS are trivial then the rule system is context-free.

**Theorem 2 (equivalence).** Consider a complete and context-free BSS defining  $\Rightarrow$  and the BSS-CRS constructed from it as in Definition 11 above defining  $\rightarrow$ , and let  $\hat{\cdot}$  denote the relation  $t \hat{\cdot} ?(t)$ . Now

$$\Rightarrow = \hat{\cdot} \cdot \rightarrow \Big|_{\Sigma}$$

*Proof.*  $\Rightarrow \subseteq \vdash \cdot \rightarrow|_{\Sigma}$  is shown by induction over the height of the proof with induction hypothesis

$$\bar{q}.Q \Rightarrow \bar{c}.C \text{ implies } ?(\bar{q}.Q) \rightarrow \bar{c}.C \quad (\text{IH})$$

- If the proof of  $\bar{q}.Q \Rightarrow \bar{c}.C$  is by an axiom then (IH) is immediate.
- If the proof of  $\bar{q}.Q \Rightarrow \bar{c}.C$  is by an inference rule with form as in the definition then there is a rewrite sequence

$$?( \bar{q}.Q ) \rightarrow !_1( \bar{q}.Q, \bar{q}_1.? ( Q_1 ) ) \quad (N_0)$$

$$\rightarrow !_1( \bar{q}.Q, \bar{c}_1.C_1 ) \quad (\text{IH})$$

$$\rightarrow !_2( \bar{q}.Q, \bar{c}_1.C_1, \bar{q}_2.? ( Q_2 ) ) \quad (N_1)$$

$$\rightarrow !_2( \bar{q}.Q, \bar{c}_1.C_1, \bar{c}_2.( C_2 ) ) \quad (\text{IH})$$

$$\vdots$$

$$\rightarrow !_n( \bar{q}.Q, \bar{c}_1.C_1, \dots, \bar{c}_{n-1}i.C_{n-1}, \bar{q}_n.? ( Q_n ) ) \quad (N_{n-1})$$

$$\rightarrow !_n( \bar{q}.Q, \bar{c}_1.C_1, \dots, \bar{c}_n.C_n ) \quad (\text{IH})$$

$$\rightarrow \bar{c}.C \quad (N_n)$$

The converse,  $\Rightarrow \supseteq \vdash \cdot \rightarrow|_{\Sigma}$ , amounts to showing that the reduction  $\rightarrow|_{\Sigma}$  can be transformed into the precise sequence of steps that would have been generated from the proof tree by the BSS-CRS construction.

- Every  $?_i$ -subterm can only be copied until it is either reduced with the appropriate ( $N$ ) or ( $N_0$ ) rule or removed. It is always possible to move or introduce (by completeness) the appropriate ( $N$ ) or ( $N_0$ ) rule right after the introduction of the  $?_i$ .
- Every  $!_i$ -subterm can only be copied until it is either reduced with the appropriate ( $N_i$ ) rule or removed. It is always possible to move or introduce (by context-freeness and completeness) the appropriate ( $N_i$ ) rule right after the elimination of the  $?_i$ -subterm in the  $!_i$ -term.

Applying these measures recursively will force the sequence of steps to be precisely as would be generated from the proof tree.  $\square$

### 4.3 It's a Wrap

To close the loop we observe without proof that

**Lemma 3.** *A BSS constructed from a CRS is complete.*

**Lemma 4.** *A BSS constructed from a CRS with no critical pairs (an “orthogonal” CRS) is context-free.*

From these we easily get the following connection.

**Corollary 1 (round trip).** *If an orthogonal CRS is used to construct a BSS and then a BSS-CRS, then the transitive reflexive closures of the original rewrite relation and the final rewrite relation restricted to the original terms are equivalent.*

## 5 Conclusions

In this paper we have seen how to derive a big step semantics corresponding to a given combinatory reduction system, thus allowing us to construct big step semantics in which evaluations are equivalent the reductions in the CRS. The validity of this method was established through constructive proofs, an outline of which was given in the paper.

Further, we discussed the reverse construction, from a given big step semantics to a combinatory reduction system.

Both of these conversions have been implemented, see [3,13] for details.

We have shown that one can convert freely between rewrite systems and big step semantics, with small restrictions on said systems. This allows one to use the formalism best suited to the task at hand, be it implementation of a programming language, a closer understanding of the system, or reasoning about properties of the system.

One significant difference between the notion of big step semantics presented here and such formalisms as natural semantics is that we allow general substitution in the rules.

### 5.1 Further Work

It would be interesting to consider ways of improving the methods presented in this paper. In particular, deriving a *normaliser* for some CRS would be of great interest. That is, can we ensure that a constructed BSS gives just the evaluations to normal forms when they exists. This may be possible if we impose some restrictions on the CRS.

Another path we have not studied is to start with a BSS1, construct the BSS1-CRS, and then construct a BSS2 from that – what then is the relationship between BSS1 and BSS2?

Finally, we would like to understand the relationship with conditional rewriting [15,7,11] as well as the most recent developments in rewriting logic [16].

## Acknowledgements

The authors would like to thank Peter Mosses for bringing us together twelve years ago, making this work possible, as well as the anonymous referees for helpful comments.

## References

1. Benaissa, Z.-E.-A., Lescanne, P., Rose, K.H.: Modeling sharing and recursion for weak reduction strategies using explicit substitution. In: Kuchen, H., Swierstra, S.D. (eds.) PLILP 1996. LNCS, vol. 1140, pp. 393–407. Springer, Heidelberg (1996)
2. Despeyroux, J.: Proof of Translation in Natural Semantics. In: Symposium on Logic in Computer Science (LICS 1986), pp. 193–205 (1986)

3. Gottliebsen, H.: Combinatory Reduction Systems and Natural Semantics. Master's thesis, DAIMI, Aarhus University (1998)
4. Kahn, G.: Natural Semantics. In: Brandenburg, F.J., Wirsing, M., Vidal-Naquet, G. (eds.) STACS 1987. LNCS, vol. 247, pp. 22–39. Springer, Heidelberg (1987)
5. Klop, J.W.: Combinatory Reduction Systems. Mathematisch Centrum, Amsterdam (1980)
6. Klop, J.W.: Term Rewriting Systems. In: Handbook of Logic in Computer Science, vol. 2, pp. 1–116. Clarendon Press, Oxford (1992)
7. Marchiori, M.: On deterministic conditional rewriting, MIT Laboratory for Computer Science, vol. 405, Computation Structures Group Memo (1997)
8. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96, 73–155 (1992)
9. Mosses, P.D.: SIS-semantics implementation system. Technical Report Daimi MD-30, Computer Science Department, Aarhus University (out of print) (1979)
10. Mosses, P.D.: Action semantics. Cambridge University Press, New York (1992)
11. Ohlebush, E.: Transforming Conditional Rewrite Systems with Extra Variables into Unconditional Systems. In: Ganzinger, H., McAllester, D., Voronkov, A. (eds.) LPAR 1999. LNCS, vol. 1705. Springer, Heidelberg (1999)
12. Plotkin, G.D.: A structural approach to operational semantics. Technical Report FN-19, DAIMI, Aarhus Universitet (1981)
13. Rose, K.H.: CRSX: Combinatory Reduction Systems with Extensions. SourceForge (2007–2009), <http://crsx.sf.net>
14. Rose, K.H.: Explicit Substitution - Tutorial & Survey. Lecture Series LS-96-3, BRICS, Aarhus Universitet (1996)
15. Rosu, G.: From Conditional to Unconditional Rewriting. In: Fiadeiro, J.L., Mosses, P.D., Orejas, F. (eds.) WADT 2004. LNCS, vol. 3423, pp. 218–233. Springer, Heidelberg (2005)
16. Serbanuta, T., Rosu, G., Meseguer, J.: A Rewriting Logic Approach to Operational Semantics. Information and Computation 207(2) (2009)

# Model-Based Testing and the UML Testing Profile

Padmanabhan Krishnan and Percy Pari-Salas

Centre for Software Assurance  
School of Information Technology  
Bond University, Gold Coast, Queensland 4229  
Australia  
`{pkrishna,pparisal}@staff.bond.edu.au`

**Abstract.** The UML Testing Profile (U2TP) provides a means of using UML for test case specification. In this work we show how the concepts of model-based testing can be mapped to U2TP at the conceptual level. We discuss structural as well as behavioural issues that allow certain aspects of model-based testing to be considered an instance of U2TP. This is achieved without insisting that model-based testing should use UML. We show how the process of using model-based testing including test case design, test generation and test execution using a tool can be captured as an instance of U2TP. The aim of this exercise is to show that model-based testing can be adopted as part of the U2TP framework, and that one does not need a different framework to adopt model-based testing in practice.

## Personal Gratitude

I (the first author) had the privilege of working with Prof. Peter Mosses as his post-doctoral fellow at the Department of Computer Science, University of Aarhus. Although the focus of our work was on Action Semantics [1], Peter had the amazing ability to use it to educate me about a wide variety of other issues such as modelling and verification. It was mainly due to his fantastic mentoring that I developed a keen interest in formal methods. Peter also impressed upon me the importance of combining formal methods with other aspects of software engineering, including building systems using appropriate design processes and the role of tools. This taught me how to wear both a practitioner's as well as a theoretician's hat.

I am indebted to Peter for his constant encouragement and support during the initial phases of my research into formal methods. Peter, congratulations on your 60th birthday and I wish you health, happiness, and success in all areas.

## 1 Introduction

Software architectures are necessary to get a handle on complexity. They specify the structure of the components that make up the system. They also describe the

interaction between the components which can be used to identify the behaviour of the system [2]. As systems get complex their testing is important. Testing is the most popular conformance checking technique and can be complex for complex systems. Hence testing frameworks based on software architectures have been defined [3]. Such testing architectures need to specify the components and behaviour of the various components that are used in the testing process.

There are various languages to express architectures (viz., ADLs) [4]. UML is a common language to express designs. UML supports the definition of profiles which can be used to customise the notation to suit particular domains. The UML Testing Profile [5] is one such example. The profile provides a means for specifying test design and execution. We show that the profile is well suited to structure the process of testing without prescribing the internal details.

Model-based testing is a particular form of testing where models are used to specify the intended behaviour of the system and the environment [6]. The focus is only on those aspects that are currently being tested. Thus tests can be derived from the model. Within model-based testing there are many approaches including the use of formal languages like Z [7], the use of state transition systems [8] and the use of function composition [9]. However, in general model-based testing does not prescribe the use of any particular modelling language. In this context, U2TP provides a framework to capture the architecture of the test design and execution but does not affect the development of the actual models.

Neto et al. [10] present a survey study that suggests the popularity of UML among developers has lead to a majority of papers in the model-based testing literature to describe approaches that are useful for UML models. Additionally, this survey claims that “many MBT approaches are demonstrated on just a subset of the UML diagrams” and suggests that the main role of UML in model-based testing is to only provide system models from which test cases are derived.

The aim of model-based testing is to automate the testing process using models and operations on models. The aim of U2TP is to define an architecture for the testing process. In this paper we show that model-based testing can fit the UML Testing Profile. This can be achieved without specifically using any of the UML diagrams as part of the model-based testing process. We also show that the testing profile, is useful for both, test case generation and execution, as well as to build a framework in which such activities are performed. So the role of the test profile is only to specify explicitly a typical architecture that can be associated with model-based testing while the model-based testing framework automates the testing process.

Test case generation and execution are two distinct phases in the model-based testing process. After a model is defined, the generation process is usually carried on by an automated tool. There are several automated tools such as TorX [11], TGV [12], and SpecExplorer [13]. Although they differ in their particular implementation details, a common abstract architecture can be defined for these tools.

More specifically, a model-based testing tool is composed by a generator, an instantiator and a test driver. The generator is the component responsible for understanding the models and generating test case descriptions that fulfill the

test objectives. The behaviour of the generator and its interface depends on the type of model it is intended to understand. The instantiator is the component responsible for generating the (concrete) test cases that will interact with the system under test (SUT). As part of its responsibilities it interacts with a data selector to extract from the data pool the data instances that will take part in the test cases. Both, the instantiator and the data selector, depend on the model to define which data instances are needed. The instantiator also depends on the SUT to know what format should be used for building the test cases.

The key contribution of this work is to illustrate the use of model based development for testing in the context of U2TP. Normally the testing profile has to import the full UML model of the SUT for elements to be tested. Also the testing profile does not specify how to carry out the testing process. By using model-based testing we show how a full UML model of the SUT is *not required*. Furthermore, we also show how a particular testing process (as defined by a specific tool) can be included in the framework. So practitioners can adopt this particular view of model-based testing as part of the meta programming support for testing. That is, U2TP provides support for both structuring the tests (discussed in Section 2) and the execution of tests (discussed in Section 3). We present various code fragments that instantiate the specifications shown in the UML2TP framework. These code fragments relate both to the model as well as the other entities that link the model to the SUT.

## 2 Structural Issues

The UML2.0 Testing Profile (U2TP) defines various concepts to support black-box testing. The main aim of this work is to show how the general model-based testing framework satisfies the key structural aspects defined in U2TP. The relevant concepts and their informal semantics and role in the process of test development and architecture are presented. The link between U2TP and model-based testing uses the taxonomy for model-based testing [14].

In U2TP the *test context* plays a central role. It is the basis for all the testing related activities. It connects the various test components, the SUT and the various test cases. In model-based testing the model plays the central role as it captures the specification of the SUT and its environment for testing. While considering test execution, a test context contains all the relevant tests. A model can also be perceived as an entity that contains all the relevant tests. One can extract different tests from the model by using different test generation algorithms. But all of them can be directly related to the model. Thus at the abstract level a model can be a test context. But not all aspects captured by test contexts can be captured by models. At a concrete stage, issues such as if a container has to be rebuilt after test execution, the number of times a test has to be run, the presence of mock objects are all part of the test context. But these are not directly relevant for model-based testing. But the key aspects of model-based test generation and execution can be captured using a test context.

In order to respect the U2TP the other entities in the model-based testing process must correspond to the relevant entities in the U2TP meta-model. Model-based testing introduces the concept of models into the testing framework. Although models are not explicitly considered in U2TP, a model can be seen as a collection of behaviours. In other words a model can be viewed as a collection of objects tagged with the *behaviour* stereotype. A behaviour defines a relation between inputs (data and state) and outputs (data and state) of any system component. Thus, one can have basic (or simple behaviours) as well as complex ones. Then, a model not only contains the basic behaviours but also the rules to combine them and compose more complex ones.

*Behaviour* is the next entity considered. It is perhaps the most important entity in the testing framework. In the U2TP context behaviours can be sequence diagrams, traces of state machine specifications etc. Such descriptions have to be ultimately related to a test case which will be interacting with the SUT when executed. In the model-based testing context there are different types of behaviours. The first type of behaviour is at the model level, the second type at the SUT level and the third type that links the behaviour at these two levels. This linking process is achieved via an *adaptor* [14]. Without an explicit link to the SUT model-based testing is not useful to a practitioner. As the model used for testing will usually be abstract, the model level behaviour could be in terms of state transition systems, function compositions or algebraic/logical specifications. The behaviour of the SUT (at the black-box level) will usually be in terms of input-output or interactive behaviour. The link between these types of behaviours is usually achieved via scripts which is a third type of behaviour. But the third type of behaviour is not always necessary as demonstrated by the AETG system [15].

Similar to behaviours there are two types of test cases. The model level test cases are directly generated from the model. The SUT level test cases have to be generated from the model level tests and the link between model level behaviours and system level behaviours. Such links are usually defined via scripts that map the actions in the model to some functionality in the implementation.

A test case (at any level) is just a behaviour (at the corresponding level) and this is reflected in the 1-1 relationship between test cases and behaviours in Figure 1. As these concepts are closely related, it is not clear why the U2TP has a separation between them. For instance there is no mention of behaviour in the work by Dai [16]. In this work we use behaviours and test cases interchangeably.

The adaptor is not only a type of behaviour but it is also an instance of *test component*. The relationship between test cases at the model and SUT levels show a dependency to the adaptor. This dependency defines a model-level relationship rather than a run-time relationship. Hence, the link at a run-time level should be specified by certain properties that are required to hold. These properties are discussed in the next section.

A *test objective* (also called selection criteria) describes the requirements used to generate the test cases. Examples include various form of coverage, reachability, specific and random values. Usually an objective is met by a number of test cases.

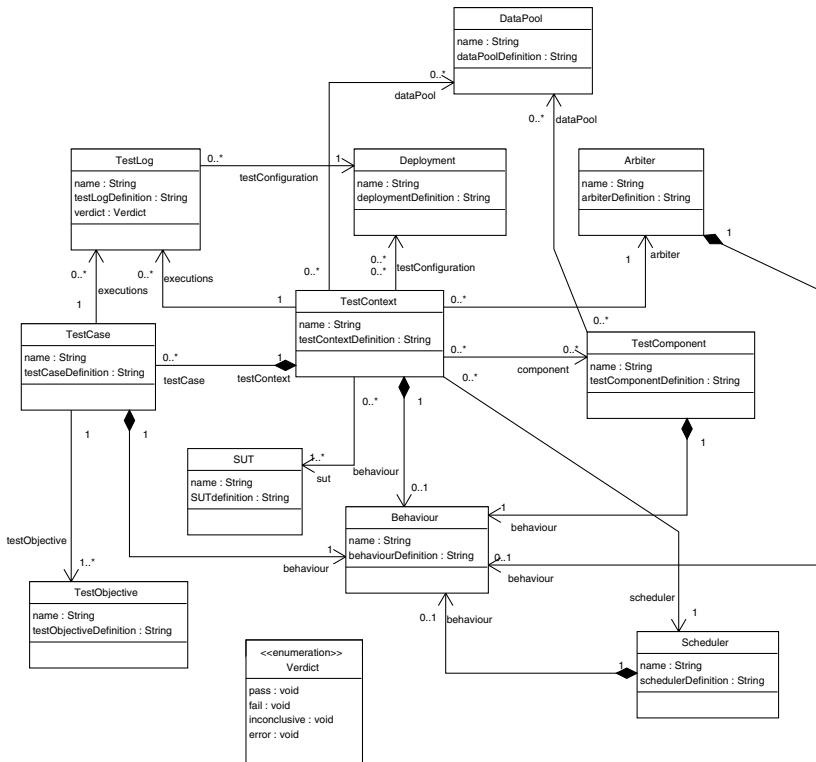
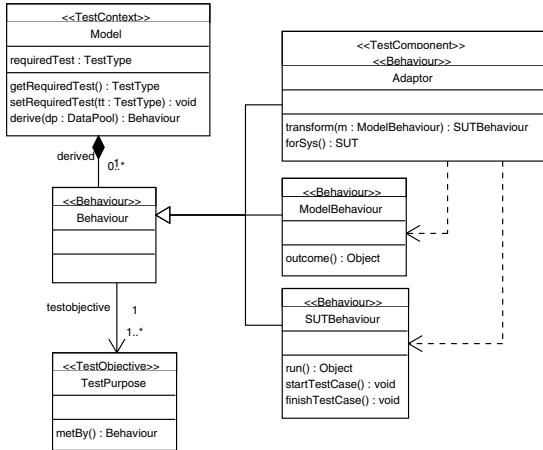


Fig. 1. Metamodel of the UML Testing Profile

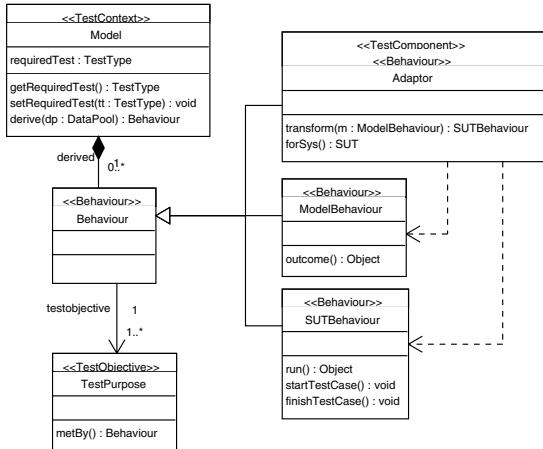
One test case may not be sufficient to meet a particular criterion and a collection of tests may be necessary. However in U2TP a particular test objective is related to one test case. This could be satisfied by concatenating different test cases with reset actions in between. Thus the concatenated test case is a single behaviour which meets a single objective. A test case can also be used to satisfy different criteria.

The *data pool* is a collection of values that is used by the various components. This has to be related to the behaviour. Without a semantic link to behaviour, a data item can be used for any behaviour which models only random testing. The test context (or model) provides the link between a derived behaviour and a particular data pool. Additionally, the *data selector* operation in U2TP is implemented by the data pool.

The final item we consider is an implementation of the *arbiter* interface. The implementation contains a validation action which is invoked by the test context to produce the verdict after the execution of a test case. The test context produces a test report which is a class with the *test log* stereotype. It contains a history of the results of all test case executions. Since it is the test context which manages the execution of the test cases it seems natural to link the test report to the test context. It is not clear why a test log is linked to a particular test case in U2TP.



**Fig. 2.** Linking MBT framework to the UML Testing Profile-I



**Fig. 3.** Linking MBT framework to UML Testing Profile-II

In Figures 2 and 3 we present a pictorial description of an model-based testing framework that show the relation between Model-Based Testing and the U2TP. Both structural aspects discussed in this section and behavioural aspects discussed in the next section are represented. Figure 2 shows the entities that describe the test cases on the model level and their link to the SUT level. The diagram captures the fact that there are three types of behaviours (one at the model level, one at the implementation level and a third that describes how the model level behaviour can be transformed to an implementation level behaviour). Figure 3 show the entities that support the generation and execution of the test cases. They also indicate some specific attributes and methods. These attributes and methods are motivated and justified in the next section.

To avoid redundancy, some aspects of U2TP, like the verdict enumeration, are not represented.

### 3 Behavioural Aspects

We now address the behaviour aspects necessary to use model-based testing. This is in terms of “What are the methods necessary to enable the use of model-based testing?” The profile does not define any functions with these classes. The necessary functions which will enable the execution of model-based testing system (including test generation and test execution) are now developed.

The relation between *test context* and *behaviour* is described by the *derive* method (called *behaviour* in U2TP). This is to indicate that the behaviours are derived from the model. In reality the behaviours that are derived also depend on the data pool. For example, if the model has the login transition, the data necessary to implement it (viz., name and password) will be associated with an instance of data pool. Thus the transition needs to be instantiated with the right data. So an object of the class data pool introduces a method (say **relevantData**) which returns the data items that are relevant to the appropriate sub-behaviour (which could be a transition or a function call). In model-based testing the data pool is initialised by synchronising it with the SUT. Thus, we introduce a method **synchronise(d)** which defines the collection of data items. After initialisation the method **relevantData** can choose the specific data element.

As many different algorithms can be applied to the model (e.g., random testing, or the Chinese postman walk) it is essential to set the type of tests that need to be derived. So it is assumed that the model has a state variable that can be set to some appropriate value. Such appropriate values cannot be defined by U2TP framework; but will be defined at the implementation level. The type **testTypes** can be an enumeration which lists all the possibilities and the attribute **requiredTest** hold a value of **testTypes**. Thus the model will need methods **m.getrequiredTest** and **m.setrequiredTest(v)** to obtain and alter the values.

The derivation of behaviours is given by the following code fragment. Let **m** be an instance of *TestContext* (i.e., a model).

```
// identify the relevant data pool list
dpList = m.getDataPool();

// identify the list of behaviours
// derive will use the relevantData method
modelBehList = m.derive(dpList);
```

The derived behaviours from the model need to be modified for testing the SUT. This is achieved by the adaptor which is a test component. It is identified by the relation *adaptor* between the test context and the test component. The adaptor, given a behaviour at the model level, will transform the model level behaviour to

a behaviour at the SUT level. This will be achieved by a method called transform associated with the adaptor class.

The code fragment is

```
// the adaptor linked to the model
ad = m.adaptor;
// transform the behaviours
sutBehList = ad.transform(modelBehList)
```

Here the SUT is mentioned explicitly. The adaptor should have a method (say `forSys`) which identifies the SUT associated with the adaptor. Note that there is no explicit relationship between the test component and the SUT. For the above system to work properly the condition `m.adaptor.forSys() == m.sut` needs to hold. That is, the SUT must match the system associated with the adaptor.

After the test cases are generated, they can be run and the results compared by the arbiter. To achieve this the SUT level behaviour should implement the *scheduler* interface defined in the U2TP profile. Every behaviour in `sutBehList` (say `bi`) can be run and the result checked against the expected value identified by the corresponding behaviour in `modelBehList` (say `b`). The methods `run` (on SUT level behaviours) and `outcome` (on model level behaviour) are assumed.

The code fragment to execute the tests and examine the results is:

```
sutResult = bi.run();
modelResult = b.outcome();
// produce the verdict
m.arbiter.compare(modelResult,sutResult);
```

The above code reflects the fact that the arbiter is not connected to the test log. Also it is not clear how the test log can have its *verdict* as it is the role of the arbiter to arrive at the verdict. In U2TP the informal description indicates that the arbiter evaluates the result of a test case or a test context. In the context of model-based testing the role of the arbiter is restricted to comparing the outcomes at the test case level.

Other aspects such as test objectives can be addressed in a similar fashion. For instance, a method (say `metBy`) that returns a list of test cases can be associated with the test objectives. One can then check if `sutBehList` is identical to `objective.metBy()`; to determine if all the relevant test cases are generated.

If all the tests in this list pass then one can claim that the test objective is satisfied.

## 4 Example

An example derived from a commercial tool is now described using the framework described earlier. SMART-MBT (from Smart Testing Technologies<sup>1</sup>) is a model-based testing tool where the state transition models are described in Prolog and

---

<sup>1</sup> [www.smarttestingtechnologies.com.au](http://www.smarttestingtechnologies.com.au)

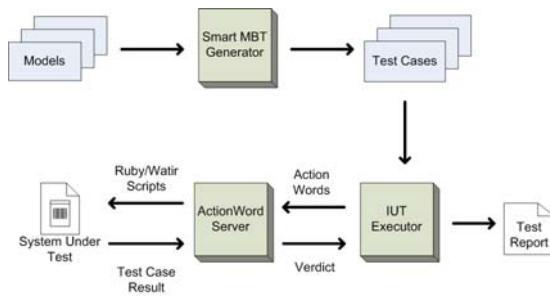


Fig. 4. Architecture of the SMART-MBT tool

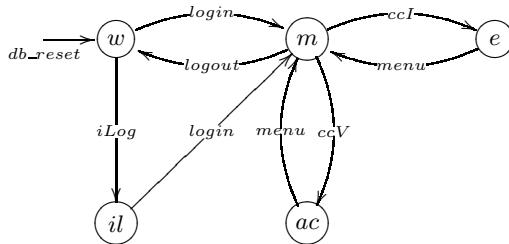


Fig. 5. Abstract model of a web system

the link to the applications is defined by scripts developed in Ruby. It uses the action word framework [17]. That is, the action on a transition is used to trigger the corresponding function in Ruby. Thus a behaviour in the model (which is a sequence of transitions in Prolog) generates a test sequence that can be used directly on the SUT. The architecture of this tool is described in an abstract way in Figure 4.

For the purposes of this example, the SUT is a web-based system administration software where users can be created and deleted by an administrator. It also registers clients to which users are associated. The administrator also assigns privileges to the user. A user can only exercise those privileges.

As the aim of the paper is mainly to show the use of U2TP in the context of defining a design process and use tools, the actual test execution is not defined in full detail. So the implementation level details of the SUT and test cases are also not necessary. Only the aspects of SMART-MBT and of the SUT that are necessary to understand the architecture of the tool and the testing process followed are described. An abstract model of the SUT is depicted in Figure 5. In this state/transition based model, transitions represent the execution of certain operations. Any operation can result in a successful or unsuccessful execution. Dotted lines represent unsuccessful executions. In both cases, the execution of any operation updates the state of the system.

For SMART-MBT the test context is a Prolog program that defines the various transitions in the model. The first step in a model-based testing process is to synchronise the model with the SUT. In this example, this synchronisation

is carried on by the transition `db_reset`. This transition sets up in the model the initial values for the system variables. It is associated to the action word `dbReset(snapshot)` where `snapshot` contains the initial values of the application's database, including, for example, the table of users.

The initial values set up in the model match those in the database. Then, after this action word is executed we can be sure that, for example, a user name that is valid for the model is also valid for the SUT. The `db_reset` transition and its action word `dbReset(snapshot)` are the implementation of method `synchronise(d)`.

The model specification of `db_reset` is shown below where the set of users and clients is specified.

```
transition(db_reset(Snapshot)) :-  
    getv(current_page, undef),  
    setv(users, [  
        [u1, admin, admin, true,  
         'Administrator', ''],  
        [u3, tester, tester, tester, true,  
         'Tester', 'tester@email.com'],  
        [u4, viewer, viewer, viewer, true,  
         'Viewer', 'viewer@email.com']  
    ]),  
    setv(clients, []),  
    setv(current_page, welcome).
```

In the following, the main functionality we consider is the creation of clients in the system. For instance, the transitions related to creating clients are shown below.

```
transition(createClientInvalid(Name,Addr,Error)) :-  
    \+clientExists(Name),  
    nameInvalid(Name),  
    CurrentPage = error  
    Error = "EC-Invalid-Name" .  
  
transition(createClientAgain(Name,Addr,Error)) :-  
    clientExists(Name),  
    CurrentPage = error  
    Error = "EC-Name-Exists" .  
  
transition(createClientValid(Name,Addr,Error)) :-  
    \+clientExists(Name),  
    nameValid(Name),  
    append(clients, [[Name,Addr]], _Clients),  
    setv(clients, _Clients),  
    Current_Page = accountCreated .
```

The first two transitions represent invalid user input while the last one represents a valid transition. The names of the transitions, viz., `createClientInvalid`,

`createClientAgain` and `createClientValid` are the action words used to link the model to the SUT. The model behaviour is defined in terms of the transitions declared in this Prolog program.

Every transition could have a data generation rule. So `createClientInvalid` has the following rule associated with it.

```
params(createClientInvalid(Name,Addr,Error)) :-  
    choose(errList,List),  
    member([Name,Addr],List).
```

where `errList` is the list of erroneous names. This rule is an implementation of the `relevantData()` method. At the model level, the data generation rules are grouped into the `DataPool` class, so that the method `m.dataPool()` identifies such data generation rules. So `m.dataPool().relevantData` will return a list of name, address pairs from `errList`. Each item in this list can now be accessed to generate a test case.

Having defined the model one can use on-line or off-line techniques to generate the tests. On-line techniques are necessary as it is not always possible to generate the entire state machine for the modelled application. The on-line techniques supported by the SMART-MBT tool allows one to perform interactive testing. The off-line techniques in the SMART-MBT tool refer to the complete coverage of all transitions in the model, or random testing up to a certain length. For all these activities one has to derive the tests from the model before they can be applied to the SUT. However, different techniques will perform the derivations in different ways. Thus, at the model level, one needs to set the type of the generated test cases. In the case of SMART-MBT the values associated with `testTypes` can be `interactive`, `chinesePostman` or `random`. The `interactive` mode specifies that the tests are generated on the fly by the tester, while the `random` mode specifies that the tests are automatically generated by randomly selecting transitions. The `chinesePostman` is a specific algorithm to obtain transition coverage.

The test derivation processes is captured by the following code.

```
// v is a suitable value  
m.setrequiredTest(v);  
m.derive(m.dataPool());
```

As the application is a web-based system, the SUT level tests are in terms of Ruby and Watir scripts. Thus `m.adaptor` identifies an object that points to these scripts. Particularly, in the case of our example, a server (called *ActionWord-Server* - see Figure 4) is configured so that it listens into a defined port for calls to the appropriate scripts from a Prolog program.

The system defined a method to check the state of the SUT and can be used to ensure that the pre and post conditions of the transitions are satisfied. The method is also used to encode the test oracle.

For each transition (i.e., an action word), there is a corresponding function in Ruby/Watir. For example, the script

```

@ie.text_field(:name,'name').set(nameVal)
@ie.text_field(:name,'address').set(addrVal)
@ie.button(:value,'Create').click
check(@ie.title = "Account Created")

```

will correspond to the `createClientValid` transition. The last line involving the checking corresponds to the `m.arbiter.compare` method.

Finally, the SMART-MBT tool contains a module that provides control over the execution of the test cases. This module implements the *SUTBehaviour* class and includes the `run()` method. It also generates the calls to the port in which *ActionWordServer* is listening. Each test case execution is logged by the *SMARTLogger* module, an implementation of the *TestReport* class.

#### 4.1 Supporting Multiple Tools

The U2TP specification [5] mentions two uses for the testing profile. The profile can be used in a stand alone fashion to specify testing system artifacts or it can be used in an integrated manner with UML for handling system and test artifacts together. As part of this research work we have explored both uses.

Using U2TP to specify the artifacts of a typical model-based testing framework allowed us to bring different testing tools to work together, not only

```

public class SMARTCatTest extends TestCase {
...
    public boolean createClient
        (String clientname, String address)
    throws Exception
    {
        ie.link(url,"/clientcreate/").click();
        ie.waitUntilReady();
        assertTrue("At create client page",
        ie.title().contains("Create Client"));

        ie.textField(name, "clientname").set(clientname);
        edit_field((IETextField)ie.textField(name, "streetaddress"),
                    address);

        ie.button(value, "Save").click();
        ie.waitUntilReady();
        assertTrue("Successful creation",
        ie.title().contains("Account Created"));
        return true;
    }
...
}

```

**Fig. 6.** Extract of the Java code implementing the Action Words

model-based tools but also “classic” testing tools. As an example we coupled SMART-MBT and JUnit by replacing the modules which execute the test cases against the SUT in the SMART-MBT tool with equivalent JUnit-based modules. Figure 6 shows an extract of these modules.

The new implementation level tests are now methods of a class that extends `jUnit.TestCase`. This methods use the Watij library (the Java port of Watir). The method `m.arbiter.compare` is now implemented by the suitable assertion methods in the JUnit framework. The adaptor `ActionWordServer` is now a Java server that listens to the port where the Prolog modules of SMART-MBT write the test cases in an XML format. The fact that test cases are passed from SMART-MBT to Ruby (or Java) in a XML format enhances replaceability of the adaptor.

The transition `dbreset` is still used to synchronise the model and the SUT. However, the use of JUnit opens the possibility of using the `setUp()` method of the `TestCase` class to perform this task. Finally, the `TestResult` class of JUnit is the actual implementation of `TestReport` replacing the `SMARTLogger`.

## 5 Conclusion

This article has shown how U2TP can be used to incorporate model-based testing in the software engineering process. The main advantage is that it provides a systematic way to develop models (not necessarily in UML) and their use in testing. Our simple example has shown that at a conceptual level a commercial MBT tool implements the U2TP profile. It has also shown that the user only needs to define certain methods and attributes to support the process, and that components developed in different platforms (or languages) can be interchangeable. The fact that JUnit, which is shown as an example implementation in the U2TP specification[5], has been used to replace some modules of the SMART-MBT tool reinforces the thesis that model-based testing can organised to be an instance of U2TP.

Furthermore we have shown that at a conceptual level behaviours can be specified in any modelling language. The behaviours are and decoupled from the actual structure and use of the testing system. So one need not use UML as the modelling tool for model-based testing to be related to U2TP. It is the structure and process of developing tests using model-based testing that is an instance of U2TP.

## Acknowledgments

The authors thank Dr. Kelvin Ross for his support and to Smart Testing Technologies for providing free access to their tools. The second author is supported by a grant from Smart Testing Technologies.

## References

1. Mosses, P.D.: Action Semantics. Tracts in Theoretical Computer Science, vol. (26). Cambridge University Press, Cambridge (1992)
2. Sitaraman, M., Atkinson, S., Kulczycki, G., Weide, B.W., Long, T.J., Bucci, P., Heym, W.D., Pike, S.M., Hollingsworth, J.E.: Reasoning about software-component behavior. In: Frakes, W.B. (ed.) ICSR 2000. LNCS, vol. 1844, pp. 266–283. Springer, Heidelberg (2000)
3. Muccini, H., Dias, M.S., Richardson, D.J.: Towards software architecture-based regression testing. In: WADS 2005: Proceedings of the workshop on Architecting dependable systems, pp. 1–7. ACM, New York (2005)
4. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering 26(1), 70–93 (2000)
5. OMG: Uml 2.0 testing profile (2005),  
<http://www.omg.org/cgi-bin/doc?formal/05-07-07>
6. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann Publishers Inc., San Francisco (2006)
7. Malik, P., Utting, M.: CZT: A framework for Z tools. In: Treharne, H., King, S., Henson, M.C., Schneider, S. (eds.) ZB 2005. LNCS, vol. 3455, pp. 65–84. Springer, Heidelberg (2005)
8. Gargantini, A.: Conformance testing. In: Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., Pretschner, A. (eds.) Model-Based Testing of Reactive Systems. LNCS, vol. 3472, pp. 87–111. Springer, Heidelberg (2005)
9. Aichernig, B.K.: Mutation Testing in the Refinement Calculus. Formal Aspects of Computing 15(2-3), 280–295 (2003)
10. Neto, A.D., Subramanyan, R., Vieira, M., Travassos, G.H., Shull, F.: Improving evidence about software technologies: A look at model-based testing. IEEE Software 25(3), 10–13 (2008)
11. Tretmans, J., Brinksma, E.: TorX: Automated model based testing. In: First European Conference on Model-Driven Software Engineering, Nuremberg, Germany (2003)
12. Jard, C., Jéron, T.: TGV: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. J. Software Tools for Technology Transfer 7(4), 297–315 (2005)
13. Campbell, C., Grieskamp, W., Nachmanson, L., Schulte, W., Tillmann, N., Veanes, M.: Model-based testing of object-oriented reactive systems with Spec Explorer. Technical Report MSR-TR-2005-59, Microsoft Research (2005)
14. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing. Technical Report 04/2006, Department of Computer Science, The University of Waikato, New Zealand (2006)
15. Cohen, D.M., Dalal, S.R., Fredman, M.L., Patton, G.C.: The AETG system: an approach to testing based on combinatorial design. IEEE Transactions on Software Engineering 23(7), 437–444 (1997)
16. Dai, Z.R.: Model-driven testing with UML 2.0. In: EWMDA: European Workshop on Model Driven Architecture, pp. 179–187 (2004)
17. Buwalda, H.: Action figures. In: Software Testing and Quality Engineering, pp. 42–47 (2003)

# A Complete, Co-inductive Syntactic Theory of Sequential Control and State<sup>\*</sup>

Kristian Støvring<sup>1</sup> and Soren B. Lassen<sup>2</sup>

<sup>1</sup> IT University of Copenhagen, Denmark  
`kss@itu.dk`

<sup>2</sup> Google Inc., Mountain View CA, USA  
`soren@google.com`

**Abstract.** We present a co-inductive syntactic theory, *eager normal form bisimilarity*, for the untyped call-by-value lambda calculus extended with continuations and mutable references.

We demonstrate that the associated bisimulation proof principle is easy to use and that it is a powerful tool for proving equivalences between recursive imperative higher-order programs.

The theory is modular in the sense that eager normal form bisimilarity for each of the calculi extended with continuations and/or mutable references is a fully abstract extension of eager normal form bisimilarity for its sub-calculi. For each calculus, we prove that eager normal form bisimilarity is a congruence and is sound with respect to contextual equivalence. Furthermore, for the calculus with both continuations and mutable references, we show that eager normal form bisimilarity is complete: it coincides with contextual equivalence.

Eager normal form bisimilarity is inspired by Böhm-tree equivalence in the pure lambda calculus. We clarify the associated proof principle by reviewing properties of Böhm trees and surveying previous work on normal form bisimulation.

## 1 Introduction

Program equivalence is a fundamental and challenging concept in programming language semantics. An important goal is to develop *useful* techniques for reasoning about *realistic* programming languages. In general, it is infeasible to reason directly about programs in the syntax of actual programming languages, because their syntax is rich and tailored for other purposes. A practical solution to this problem is to use a semantic description framework such as action semantics [43] to specify a direct or component-based [42] mapping from the programming language syntax to a semantic description language equipped with a formal semantics and effective reasoning techniques. This mapping should allow the conceptual analysis of programming language constructs to be expressed directly, avoiding elaborate and obscure encodings, and therefore the semantic

---

\* Extended version of an earlier conference paper [58], incorporating parts of Chapter 2 of the first author's PhD dissertation [57].

description language must express the fundamental computational concepts of the programming language [44].

Our aim is to contribute to the development of effective reasoning techniques for semantic description languages that combine multiple computational concepts. In this paper we address the fundamental higher-order and imperative concepts of sequential functional and object-oriented programming languages. This is an active area of research and new and better frameworks and techniques for reasoning about program equivalence are continually being developed. Nonetheless, there are still no general and easy to use methods that capture the features and subtleties of actual programs in languages that combine general recursion, higher-order functions and objects, mutable state, and non-local control flow.

Denotational semantics and domain theory cover many programming language features but straightforward models fail to capture certain important aspects of program equivalence, especially concerning mutable state. The solutions to these “full abstraction” problems, including game semantics, are complex.

Syntactic reduction calculi and equational theories are easy to use but they exclude many important program equivalences.

The broadest notion of program equivalence is Morris-style contextual equivalence which equates two terms if they behave the same in all program contexts. The quantification over all program contexts makes it impractical to use the definition directly to prove programs contextually equivalent.

Syntactic methods based on operational semantics—context lemmas, applicative bisimulation, and operationally-based logical relations—generally incur modest “mathematical overhead” and are easy to use for certain classes of program equivalences. For instance, applicative bisimulation is very useful for proving the equivalence of programs that output infinite data structures. Moreover, many of these syntactic methods are complete for proving contextual equivalences in effect-free calculi. However, all these proof principles are weak for program equivalences involving general higher-order functions because, somewhat like the definition of contextual equivalence, they involve universal quantifications over all continuations, stores, and/or function arguments.

For example, fixed-point combinators are higher-order functions that make essential use of higher-order arguments. What does it take to prove the equivalence of two different fixed-point combinators? A proof obligation that involves a universal quantification over all possible arguments to the fixed-point combinators is about as difficult as proving that the fixed-point combinators are contextually equivalent from first principles.

This example is easily solved using a different class of syntactic theories which originate from the theories of Böhm tree equivalence and Lévy–Longo tree equivalence. They can be presented as bisimulation theories, called *normal form bisimulation*, without explicit reference to trees. Sangiorgi originally introduced this bisimulation characterisation of Lévy–Longo tree equivalence for the lazy  $\lambda$ -calculus under the name “open applicative bisimulation” [53, 54] which is in turn derived from trigger bisimulation for higher-order concurrency [51].

Normal form bisimulation is based on symbolic evaluation of open terms to normal forms. It does not involve any universal quantification over function arguments and is therefore, in some respects, a more powerful proof principle for proving equivalences between recursive higher-order functions than other operationally-based syntactic methods. However, normal form bisimulation has only been developed for state-less  $\lambda$ -calculi and is in general not fully abstract, i.e., not complete with respect to contextual equivalence.

In this article we first review properties of Böhm trees and survey previous work on normal form bisimulation. With that background, we then address the shortcomings of normal form bisimulation mentioned above by presenting syntactic bisimulation theories for the untyped call-by-value  $\lambda$ -calculus extended with continuations and/or mutable references. The theories all extend eager normal form (enf) bisimulation, a variant of normal form bisimulation for the pure call-by-value  $\lambda$ -calculus [30].

The extension with continuations is for an untyped call-by-value version of Parigot's  $\lambda\mu$ -calculus [45]. This extension is based on the second author's normal form bisimulation theory for the untyped  $\lambda\mu$ -calculus [32], but here we consider call-by-value reduction instead of head reduction.

The extension with mutable references is for a calculus that is essentially Felleisen and Hieb's stateful  $\lambda$ -calculus [15]. However, their “ $\rho$ -application” is a primitive in our calculus; hence we name it the  $\lambda\rho$ -calculus. In this extension, a bisimulation is a *set* of binary relations on terms, not a single binary relation on terms. The idea is that the set of related terms grows monotonously as “time passes”: related terms are required to stay related when the states change. This “relation-sets bisimulation” approach is adapted from bisimulation theories for imperative calculi [25, 29] and existential types [61].

Finally, we extend the two theories above to a combined  $\lambda\mu\rho$ -calculus which is essentially a variant of Felleisen's  $\lambda_v$ -CS calculus for Scheme [14].

The resulting bisimulation proof principle for proving semantical equivalences between terms inherits the best properties of normal form bisimulation and relation-sets bisimulation, namely:

- Like other kinds of normal form bisimulation, the enf bisimulation proof obligations for continuations and mutable references require no universal quantifications over function arguments or continuations or stores.
- The relation-set structure represents the “possible worlds” necessary to capture the behaviour of mutable references.

We demonstrate the power and ease of use of the resulting enf bisimulation proof principle for continuations and mutable references by proving the correctness of Friedman and Haynes's encoding of `call/cc` in terms of “one-shot” continuations [16]. Despite the subtlety of their encoding and the mix of higher-order functions, first-class continuations, and mutable references, the bisimulation proof is remarkably straightforward.

The enf bisimulation theories for the pure  $\lambda$ -calculus and the extensions with continuations and/or mutable references are modular: enf bisimilarity for each of the extended calculi is a fully abstract extension of enf bisimilarity for its

sub-calculi. This is similar to the relationship between Felleisen and Hieb’s syntactic theories for control and state [15] but contrasts the situation for contextual equivalence because each language extension makes contextual equivalence more discriminative on terms of the sub-calculi.

One of the main technical contributions of the work behind this article is a proof that *enf* bisimilarity for the calculus extended with continuations and/or mutable references is a congruence. As an immediate consequence of congruence, *enf* bisimilarity is included in contextual equivalence for each calculus. For the pure  $\lambda$ -calculus as well as the two extensions with only continuations and only mutable references, *enf* bisimilarity is strictly smaller than contextual equivalence, that is, *enf* bisimulation is a sound but incomplete method for proving contextual equivalence. However, for the full calculus with both continuations and mutable references, we prove that *enf* bisimilarity is fully abstract in the sense that it coincides with contextual equivalence.

In summary, we present a complete, co-inductive syntactic theory for a calculus with higher-order functions, continuations, and mutable references, and we demonstrate the power and ease-of-use of the bisimulation proof method for proving equivalences between recursive programs. Our results provide further illustration of the promise of normal form bisimulation as a basis for syntactic theories and proof principles, demonstrated by results for other pure and extended  $\lambda$ -calculi in the literature [31, 32, 34–36, 54].

## 1.1 Limitations

Although our theory for the combined  $\lambda\mu\rho$ -calculus captures key functional and imperative aspects of the programming language Scheme, it lacks constants such as `nil`, `cons`, numerals, and arithmetic operators. These constants need to be encoded in our calculus, e.g., using standard  $\lambda$ -calculus encodings [7], but such encodings are in general not faithful to the constants’ equational properties. For instance, addition of values should be commutative, up to contextual equivalence—that is, the representations of the Scheme terms `(lambda (x y) (+ x y))` and `(lambda (x y) (+ y x))` in the  $\lambda\mu\rho$ -calculus should be equivalent—but this fails for encodings of arithmetic in the  $\lambda\mu\rho$ -calculus. One can nevertheless use the results in this article to prove interesting equivalences between terms with encoded constants; see for example Section 8.

There does not seem to be a satisfactory direct definition of normal form bisimulation (or Böhm-tree equivalence) for untyped calculi with constants. One simple reason is that normal form bisimulation is based on symbolic evaluation of open terms: it is not at all clear how a constant such as `+` should behave when one (or both) of its arguments is a free variable. In particular, it is not clear how to define a general notion of reduction for open, untyped terms with constants such that the open terms  $x + y$  and  $y + x$  become bisimilar. More abstractly, the idea behind normal form bisimulation is to treat function arguments as “black boxes” that must be used in lockstep by bisimilar functions. While this is appropriate for function arguments that are themselves functions—as is always the case in our untyped calculi—it is not appropriate for function arguments that belong

to an inductive data type such as natural numbers. To prove properties about such arguments, one often needs to use structural induction, and therefore such arguments should not be treated as black boxes.

Lassen and Levy [35, 36] have recently extended eager normal form bisimulation to a calculus with types, and in particular recursive types and universal types, but not effects. In their calculus, one can for example use a recursive type to encode natural numbers such that the expected elementary identities hold. It remains to investigate if eager normal form bisimulation can be adapted to a calculus with both types and control and state effects.

Unlike in some of the related work described in the next section [3, 25, 27, 60] the mutable references we consider are not first-class. We believe that a good treatment of normal form bisimulation for first-class references needs to be for a typed calculus.

## 1.2 Related Work

There exists a large body of work on syntactic theories and semantic models (domains and games) for  $\lambda$ -calculi with continuations and mutable references. We only survey a few works on syntactic theories most closely related to the results in this article.

As mentioned in the introduction, our results build directly on recent work on normal form bisimulation for call-by-value [30] and the  $\lambda\mu$ -calculus [32] and on relation-sets bisimulation for existential types [61] and untyped imperative  $\lambda$ -calculus [25, 29].

One particular inspiration for the work presented in this article is the seminal research by Felleisen *et al.* on syntactic theories for sequential control and state [15]. The calculi in *op.cit.* are enriched with constants and  $\delta$ -reduction but otherwise the state calculus is essentially what we call the  $\lambda\rho$ -calculus in this article. The control calculus differs from the  $\lambda\mu$ -calculus but they are comparable. (Their relationship is analyzed by de Groote [12] and by Ariola and Herbelin [6]. We found that it was easiest to define eager reduction on open terms, enf<sub>s</sub>, and enf bisimilarity for the  $\lambda\mu$ -calculus.) The syntactic theories of successive  $\lambda$ -calculus extensions by Felleisen et al. [15] are modular (conservative extensions), like our syntactic theories. An important difference is that the syntactic theories in *op.cit.* are *inductive* in the sense that all equations are derived inductively from equational axioms and inference rules, whereas our bisimulation theories are *co-inductive* and therefore equate many more programs.

Another body of related work is Mason and Talcott’s CIU (“closed instantiations of uses”) characterizations of contextual equivalence for functional languages with mutable references and continuations [38, 62]. (The context lemmas for the  $\lambda\mu$ -calculus by Bierman [8] and by David and Py [11] are essentially CIU characterizations.) The CIU equivalences are complete syntactic theories but the resulting proof methods are in many cases weaker than normal form bisimulation.

Most co-inductive syntactic programming language theories in the literature are variants and extensions of Abramsky’s applicative bisimulation [1]. Applicative

bisimulation is fully abstract for effect-free functional calculi [1, 13, 46] but not for  $\lambda$ -calculi with continuations and/or mutable references.

Ritter and Pitts [50] define a form of applicative bisimilarity for a functional language with mutable references. It is sound but not complete. In fact, it does not equate many of the well-known, subtle contextual equivalences between programs with state [40].

Wand and Sullivan [64] define a CPS language with mutable references and show that applicative bisimilarity is both sound and complete. They use the CPS language as a semantic meta-language and CPS translate a source language with state into the CPS language. But they do not give an independent characterization of the induced syntactic theory on source terms via the CPS transform.

Koutavas and Wand’s relation-sets bisimulation theory [25] is complete for a general “direct-style” imperative calculus. However, it involves a universal quantification over closed function arguments, unlike our normal form bisimulation theories. The technique has additionally been used for an untyped imperative object-oriented calculus [23] and for a typed and class-based object-oriented language [24]. In more recent work, Sangiorgi et al. [55] develop relations-sets bisimulation principles for different higher-order calculi, including an untyped calculus with state. Sumii [60] gives a complete relation-sets bisimulation principle for a calculus with universal and existential types as well as state. Sumii [59] generalizes relation-sets bisimulation in order to reason about more general contextual properties, including contextual equivalence, space improvement, and memory safety. All of the proof principles above involve quantification over closed function arguments.

Merro and Biasi [39] present a complete bisimulation theory for a CPS calculus. It can be viewed as a kind of applicative bisimulation, presented as a labelled transition system in the style of Plotkin (see Gordon [17]), and also leads to a context lemma.

Pitts and Stark [47, 48] develop syntactic theories based on operationally-based logical relations that address many of the subtleties of contextual equivalences between programs with mutable references. The relation-sets bisimulation theories for mutable state, in general, are alternative approaches with a very different meta-theory. For logical relations the key proof obligation is existence, whereas the key proof obligation for the bisimulation theories is congruence.

Ahmed et al. [3] present a step-indexed [4, 5] logical-relations proof principle for a stateful calculus with universal types, existential types, recursive types, and reference types. The proof principle allows one to prove sophisticated equivalences between typed, stateful programs, but does not cover control effects. Vytiniotis and Koutavas [63] relate the step-indexing approach to relation-sets bisimulation in the setting of a pure calculus with recursive types.

In recent work, independent from the work presented in this article, Jagadeesan et al. [21] present a bisimulation proof principle for an aspect-oriented calculus. The proof principle, inspired by open bisimulation and eager normal form bisimulation, allows one to prove equivalences between stateful programs

by means of an encoding of references using aspects. The technique does not cover non-local control flow due to the maintenance of a “stack discipline” in the definition of a bisimulation, and as a result, the proof principle is complete for a calculus without control effects.<sup>1</sup> On a more technical note, bisimulations are defined by means of a labelled transition system (LTS), resulting in both a congruence proof and a proof principle in a different style than ours. In other work, Laird [27] presents a LTS for a calculus with types and state effects, but not control effects. The resulting bisimulation proof principle is reminiscent of eager normal form bisimulation.

Finally, we note that the modularity of the enf bisimilarity theories for control and state resembles the modularity of game semantics for control and state [2, 26]. We expect that this reflects a deeper connection between normal form bisimulation and game semantics. This connection is suggested in the aforementioned related works on LTSs [21, 27] and typed normal form bisimulation [35, 36]. We discuss it briefly in Section 10.1.

## 2 Böhm Trees and Normal Form Bisimulation

Normal form bisimulation is a general name for a number of equivalence proof methods for different calculi [30–32, 34–37, 54]. These methods are inspired by the  $\lambda$ -calculus concept of equivalence of Böhm trees [7, Chapter 10] and by Sangiorgi’s “trigger”, “normal”, and “open” bisimulation for higher-order and name-passing process calculi [51, 52]. The key feature of normal form bisimulation as a proof method is that one never needs to quantify over all possible contexts of a term, or over all possible arguments to a function.

In this section we review properties of Böhm trees and survey some of the second author’s line of work on normal form bisimulation.

### 2.1 Böhm Trees

We briefly review some basic definitions and properties of Böhm trees [7, Chapters 10 and 16]. Consider the terms of the pure, untyped  $\lambda$ -calculus:

$$M ::= x \mid \lambda x.M \mid M_1 M_2$$

Every such term  $M$  either has the form  $\lambda x_1 \dots \lambda x_n. x M_1 \dots M_k$  ( $M$  is a *head normal form*) or the form  $\lambda x_1 \dots \lambda x_n. (\lambda x.N) N' M_1 \dots M_k$  ( $M$  has *head redex*  $(\lambda x.N) N'$ ). The numbers  $n$  and  $k$  can be 0. A *head reduction* is a single contraction of a head redex:

$$\lambda x_1 \dots \lambda x_n. (\lambda x.N) N' M_1 \dots M_k \xrightarrow{h} \lambda x_1 \dots \lambda x_n. N[N'/x] M_1 \dots M_k .$$

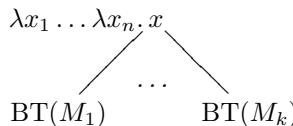
---

<sup>1</sup> More accurately, only a “signalling” operator, which resembles an operator for aborting the computation, must be added in order to obtain completeness. More powerful control operators are not needed for this purpose.

Head reduction is deterministic: if a term  $M$  is not in head normal form, then there is a unique term  $M'$  such that  $M \rightarrow_h M'$ . Say that  $M$  has a head normal form if it head-reduces to a head normal form in a number of steps. Otherwise there is an infinite reduction sequence  $M \rightarrow_h M_1 \rightarrow_h M_2 \rightarrow \dots$ . (A term head-reduces to a head normal form if and only if it is  $\beta$ -equivalent to a head normal form [7, Theorem 8.3.11], but we will not need that fact.)

**Definition 2.1.** *The Böhm tree of a term  $M$ , written  $\text{BT}(M)$ , is a potentially infinite ordered tree whose nodes and leaves are labeled with prefixes  $\lambda x_1 \dots \lambda x_n. x$  of head normal forms or with the special symbol  $\perp$ . It is defined co-inductively [20, 34] as follows:*

- If  $M$  has no head normal form, then  $\text{BT}(M)$  is the single leaf  $\perp$ .
- If  $M \rightarrow_h^* \lambda x_1 \dots \lambda x_n. x M_1 \dots M_k$ , then  $\text{BT}(M)$  is the tree with a root node labeled  $\lambda x_1 \dots \lambda x_n. x$  and subtrees  $\text{BT}(M_1), \dots, \text{BT}(M_k)$  (in that order):



Böhm trees are identified up to  $\alpha$ -equivalence in the natural way.

**Equivalence of Böhm trees.** The Böhm tree  $\text{BT}(M)$  of a term  $M$  can be viewed as a kind of infinite normal form of  $M$  (where all terms having no head normal form are identified). Therefore it makes sense to investigate what it means for two terms to have the same Böhm tree.

Let  $\approx$  be the relation of *Böhm tree equivalence*:  $M \approx M'$  if the terms  $M$  and  $M'$  have the same Böhm tree. For our purposes, the most important property of Böhm tree equivalence is:

**Proposition 2.2.** *Böhm tree equivalence is a  $\lambda$ -theory:*

1. If  $M =_{\beta} M'$ , then  $M \approx M'$ .
2. If  $M_1 \approx M'_1$  and  $M_2 \approx M'_2$ , then  $\lambda x. M_1 \approx \lambda x. M'_1$  and  $M_1 M_2 \approx M'_1 M'_2$ .

Part 2 of the proposition states that Böhm tree equivalence is a *congruence*. The proposition (and the next one) is shown in Barendregt [7, Prop. 16.4.2], as a consequence of “continuity of application,” a result about a topology on  $\lambda$ -terms that is induced by Böhm trees. A direct proof of the congruence property is also possible (see Section 2.2).

Now say that two terms  $M$  and  $M'$  are *solvably equivalent*, written  $M \sim_s M'$ , if for every term context  $C$  the (possibly open) terms  $C[M]$  and  $C[M']$  either both have a head normal form or both have no head normal form. It follows immediately from the previous proposition and the definition of Böhm trees that Böhm tree equivalence implies solvable equivalence.

**Proposition 2.3.** *If  $M \approx M'$ , then  $M \sim_s M'$ .*

The converse does not hold; we return to this issue in Section 2.2.

*Example: fixed-point combinators.* Say that a closed term  $M$  is a *fixed-point combinator* if  $Mx =_{\beta} x(Mx)$ . We can now show the well-known result that all fixed-point combinators have a common Böhm tree. Since fixed-point combinators do exist, this means that there is exactly one fixed-point combinator up to Böhm tree equivalence.

Assume that  $Mx =_{\beta} x(Mx)$ . By part 1 of Proposition 2.2,  $\text{BT}(Mx) = \text{BT}(x(Mx))$ , which means that the Böhm tree of  $Mx$  is

$$\begin{array}{ccccccc} \text{BT}(Mx) & = & x & = & x & = & x \\ & & | & & | & & | \\ & & \text{BT}(Mx) & & x & & x \\ & & | & & | & & | \\ & & \text{BT}(Mx) & & x & & x \\ & & & & | & & | \\ & & & & \vdots & & \vdots \end{array}$$

Since the Böhm tree of  $Mx$  is not  $\perp$ , the term  $Mx$  must head reduce to a head normal form. But then  $M$  must also head reduce to a head normal form. Since  $M$  is closed and  $\beta$ -equivalent to its head normal form,  $M =_{\beta} \lambda x.N$  for some  $N$ . Hence  $Mx =_{\beta} N$ , and Proposition 2.2 gives that

$$\begin{array}{ccc} \text{BT}(M) = \text{BT}(\lambda x.N) = & \lambda x.x & = \lambda x.x \\ & | & | \\ & \text{BT}(Mx) & x \\ & | & | \\ & \vdots & \vdots \end{array}$$

**Böhm trees and models of the  $\lambda$ -calculus.** Böhm trees have been used to study the theories induced by models of the  $\lambda$ -calculus, i.e., to study which closed  $\lambda$ -terms these models identify. Plotkin's model  $P_\omega$  identifies exactly those closed terms that have the same Böhm tree [7, Theorem 19.1.19], while Scott's model  $D_\infty$  identifies exactly those closed terms that have the same Böhm tree up to possibly infinite  $\eta$ -expansion [7, Theorem 19.2.9] (see Section 2.2).

## 2.2 Head Normal Form Bisimulation

We now revisit the concept of Böhm trees in a slightly different light. Let us consider the pure  $\lambda$ -calculus as a simple programming language, with an operational semantics given by head reduction. As usual in semantics, say that a *program* in this language is a closed term. Say that a term *converges* if it head-reduces to a head normal form; otherwise it *diverges*. All we will observe about a program is whether it converges or not: two programs *have the same result* if both converge or both diverge. More sophisticated observation models leading to the same notion of contextual equivalence are also possible.<sup>2</sup> Contextual equivalence is the

---

<sup>2</sup> For example, if a program converges, then one could further observe the de Bruijn index of the head variable of its result. This observation model allows one to program every recursive number-theoretic function [7, Theorem 6.3.13], as one can, e.g., define a term that converts the Church numeral  $[n]$  to a head normal form whose head variable has de Bruijn index  $n$ .

same as solvable equivalence: every open term can be converted to a closed one by  $\lambda$ -abstracting all free variables, and this does not change whether the term has a head normal form or not.

The key point is now that Propositions 2.2 and 2.3 state that Böhm tree equivalence is a congruence, and that it is therefore sound for reasoning about contextual equivalence. In other words, if two terms have the same Böhm tree, then they are contextually equivalent in the sense above.

When using Böhm tree equivalence as a proof principle for showing contextual equivalence, the concept of two terms “having the same behavior” (having the same Böhm tree) becomes more central than the actual concept of “behavior” (Böhm tree). Lassen [34] characterizes Böhm tree equivalence by means of what is called a head normal form bisimulation:

**Definition 2.4.** *A binary relation  $R$  on open  $\lambda$ -terms is a head normal form bisimulation (hnf bisimulation) if for every  $(M, M') \in R$ , one of the following conditions hold:*

1.  $M$  and  $M'$  both diverge.
2.  $M \xrightarrow{h}^* \lambda x_1 \dots \lambda x_n. x M_1 \dots M_k$  and  $M' \xrightarrow{h}^* \lambda x_1 \dots \lambda x_n. x M'_1 \dots M'_k$ , where  $(M_i, M'_i) \in R$  for all  $1 \leq i \leq k$ .

It follows immediately from the definition that head normal form bisimulations are closed under union: the union of an arbitrary family of head normal form bisimulations is itself a head normal form bisimulation. In particular, the union of all head normal form bisimulations is the greatest head normal form bisimulation. This greatest head normal form bisimulation, called *head normal form bisimilarity*, is, almost by definition, the same relation as Böhm tree equivalence. The bisimulation proof principle associated with the definition is then the following: In order to show that two terms are head normal form bisimilar (have the same Böhm tree), it suffices to construct a head normal form bisimulation relating the two terms.

The bisimulation viewpoint of Böhm tree equivalence enables a direct, relational proof that Böhm tree equivalence is a congruence [34], and hence sound with respect to contextual equivalence. It also allows for an elegant, co-inductive formulation of a proof that Böhm tree equivalence up to possibly infinite  $\eta$ -expansion is complete with respect to contextual equivalence [32] (see the next section).

**Extensional head normal form bisimulation.** Head normal form bisimulation is not a complete proof principle with respect to contextual equivalence. In other words, there are contextually equivalent terms which are not head normal form bisimilar.

The simplest example is the failure of the  $\eta$ -axiom: it follows directly from the definition that no head normal form bisimulation can relate the terms  $x$  and  $\lambda y. x y$ . On the other hand,  $x$  and  $\lambda y. x y$  are contextually equivalent, as we will see below.

Lassen [34] defines an *extensional* head normal form bisimulation as follows. Let  $\text{Rel}$  be the set of binary relations on open  $\lambda$ -terms. For every relation  $R \in \text{Rel}$ , define a relation  $H(R)$  on head normal forms: two head normal forms  $M_0$  and  $M'_0$  are related by  $H(R)$  if one of the following conditions hold.

1.  $M_0 = \lambda x_1 \dots \lambda x_n. x M_1 \dots M_k$  and  $M'_0 = \lambda x_1 \dots \lambda x_{n+m}. x M'_1 \dots M'_{k+m}$ , where  $x_{n+1}, \dots, x_{n+m}$  are not free in  $x M_1 \dots M_k$ , and  $(M_i, M'_i) \in R$  for  $1 \leq i \leq k$ , and  $(x_{n+j}, M'_{k+j}) \in R$  for  $1 \leq j \leq m$ .
2. The symmetric case where  $M_0$  begins with more  $\lambda$ 's than  $M'_0$ .

For example, if  $(y, y) \in R$ , then  $(x, \lambda y. x y) \in H(R)$ . Intuitively, the relation  $H(R)$  describes head normal forms that “match” with respect to  $R$ . Matching head normal forms are almost the same as for “non-extensional” head normal form bisimulation, expect that one is allowed to perform a number of top-level  $\eta$ -expansions on one of the two terms.

**Definition 2.5.** A relation  $R \in \text{Rel}$  is an *extensional head normal form bisimulation (extensional hnf bisimulation)* if for every  $(M, M') \in R$ , one of the following conditions hold:

1.  $M$  and  $M'$  both diverge.
2.  $M \xrightarrow{h}^* M_0$  and  $M' \xrightarrow{h}^* M'_0$ , where  $M_0$  and  $M'_0$  are head normal forms such that  $(M_0, M'_0) \in H(R)$ .

The greatest extensional head normal form bisimulation, written  $\approx_\eta$ , is called *extensional head normal form bisimilarity*. It is the same relation as Böhm tree equivalence up to possibly infinite  $\eta$ -expansion [7, p. 240]. Propositions 2.2 and 2.3 also hold for  $\approx_\eta$ , so extensional head normal form bisimulation is a sound proof principle with respect to contextual equivalence [7, 34]. Furthermore, it is easy to see that the  $\eta$ -axiom is satisfied:  $M \approx_\eta \lambda x. M x$  if  $x$  is not free in  $M$ . Therefore, extensional head normal form bisimilarity is an extensional  $\lambda$ -theory [7, p. 79].

*Example: infinite  $\eta$ -expansion.* As an example [7, 34], consider Wadsworth’s “infinite  $\eta$ -expansion” combinator  $J = \Theta(\lambda f. \lambda x. \lambda y. x(f y))$  where  $\Theta$  is Turing’s fixed-point combinator.<sup>3</sup> By construction,  $J x \xrightarrow{h}^* \lambda y. x(J y)$ . Let us show that  $J \approx_\eta I$  where  $I = \lambda x. x$  is the identity combinator. For this, we construct an extensional head normal form bisimulation relating  $J$  and  $I$ . We start with the singleton relation  $\{(J, I)\}$  and iteratively add pairs of terms in order to satisfy the definition of a bisimulation. This is done completely mechanically, by carrying out a number of head reductions and  $\eta$ -expansions. First,  $J \xrightarrow{h}^* \lambda x. \lambda y. x(J y)$  and  $I \xrightarrow{h}^* \lambda x. x$ . Since  $\lambda x. x$  has “one less  $\lambda$ ” than  $\lambda x. \lambda y. x(J y)$ , we  $\eta$ -expand it to  $\lambda x. \lambda y. x y$ . We must now add the pair of terms  $(J y, y)$  to the relation. Then we continue by reducing the terms in this new pair:  $J y \xrightarrow{h}^* \lambda z. y(J z)$  and  $y \xrightarrow{h}^* y$ . We  $\eta$ -expand  $y$  to  $\lambda z. y z$ , and must now add the pair of terms  $(J z, z)$

---

<sup>3</sup> Defined as  $\Theta = (\lambda x \lambda f. f(x x f))(\lambda x \lambda f. f(x x f))$ , and satisfying that  $\Theta f \xrightarrow{h}^* f(\Theta f)$ .

to the relation. At this point we see the pattern: we construct the extensional head normal form bisimulation

$$\{(J, I)\} \cup \{(Jx, x) \mid x \text{ is a variable}\}.$$

*Completeness.* For  $\approx_\eta$ , the converse of Proposition 2.3 holds, and therefore extensional head normal form bisimulation is a complete proof principle with respect to contextual equivalence. This can be shown using the so-called Böhm out technique [7, Section 10.3]. In terms of bisimulations, the task is to show that contextual equivalence is an extensional head normal form bisimulation [34].

As a simple example of the Böhm-out technique, consider the two terms  $M = \lambda x. x I (x II)$  and  $M' = \lambda x. x I (x \Omega I)$  where  $\Omega$  is an arbitrary diverging term. It is easy to see that  $M \not\approx_\eta M'$  since  $I \not\approx_\eta \Omega$ . Therefore, since  $\approx_\eta$  is the same relation as contextual equivalence, there must be some context  $C$  such that  $C[M]$  converges and  $C[M']$  diverges (or the other way around). Intuitively, the terms  $M$  and  $M'$  should be applied to a function  $N$  that first selects its second argument, and then selects its first argument, but this is impossible in a pure language. Fortunately there is a work-around:  $N$  simply returns all its arguments in a Church-encoded tuple:  $N = \lambda y_1. \lambda y_2. \lambda z. z y_1 y_2$ . Now define “projection” functions  $P_1 = \lambda y. y (\lambda x_1. \lambda x_2. x_1)$  and  $P_2 = \lambda y. y (\lambda x_1. \lambda x_2. x_2)$ , and let  $C = P_1 (P_2 ([] N))$ . Then  $C[M]$  converges while  $C[M']$  diverges.

In Section 11 we consider a similar completeness proof for a calculus with state. In such a calculus, examples such as the one above become conceptually simpler, since it is possible for a function to do different things each time it is called.

### 2.3 Eager Normal Form Bisimulation

In 2005, Lassen [30] presented a normal form bisimulation principle for the call-by-value  $\lambda$ -calculus, called *eager normal form bisimulation* (enf bisimulation).<sup>4</sup>

Consider a variant of the call-by-value  $\lambda$ -calculus [49] where intermediate results are explicitly named:

```
VARIABLES x, y, z
VALUES v ::= x | λx.t
TERMS t ::= v | let x=t1 in t2 | v1 v2
```

A call-by-value reduction strategy is defined by means of evaluation contexts:

```
EVALUATION CONTEXTS E ::= [] | E[let x=[] in t]
```

$$\begin{aligned} E[\text{let } x=v \text{ in } t] &\mapsto E[t[v/x]] \\ E[(\lambda x.t) v] &\mapsto E[t[v/x]] \end{aligned}$$

(Reduction is deterministic since each non-value term can be uniquely decomposed into an evaluation context and a “potential redex” [10].) Notice that reduction is defined on open terms. A normal form with respect to reduction, here

---

<sup>4</sup> The article [30] also contains an extensional variant that we discuss in Section 2.2.

called an *eager normal form*, is therefore either a value or a “call to an unknown function”:

EAGER NORMAL FORMS (ENFS)  $e ::= v \mid E[xv]$

Contextual equivalence is defined as for the pure  $\lambda$ -calculus: we only observe whether a program converges or not.

A complete definition of eager normal form bisimulation is presented in Section 3, so for now we only sketch the rest of the definition informally. First we consider a variant that does not validate Plotkin’s  $\eta_v$ -axiom [49].

Suppose we want to show that two terms  $t$  and  $t'$  are eager normal form bisimilar. As in the example in Section 2.2 we start from the singleton relation  $\{(t, t')\}$  and iteratively add more pairs of terms to the relation as required (see below). First, the two terms  $t$  and  $t'$  must either both diverge, or they must reduce to matching eager normal forms:

- Values  $v$  and  $v'$  match if either  $v$  and  $v'$  are the same variable, or  $v = \lambda x.t$  and  $v' = \lambda x.t'$  where the terms  $t$  and  $t'$  are related.
- Non-value eager normal forms  $E[xv]$  and  $E'[xv']$  match if the evaluation contexts  $E$  and  $E'$  match and the values  $v$  and  $v'$  match. Notice that the two terms must have the same variable in function call position.
- Evaluation contexts  $E$  and  $E'$  match if the terms  $E[z]$  and  $E'[z]$  are related for some fresh variable  $z$ .

Intuitively, two eager normal forms match if they consist of related “components”. As a consequence, if one wants to show that two functions  $\lambda x.t$  and  $\lambda x.t'$  are bisimilar, one proceeds by reducing the *bodies* of the two functions, and not by, e.g., considering the applications of the two functions to all possible closed arguments. In general, this approach makes it necessary to define reduction on open terms, as well as a notion of open normal form.

The idea of decomposing normal forms constitutes the core of the various normal form bisimulation principles in the literature. In the case of head normal form bisimulation, two head normal forms  $\lambda x_1 \dots \lambda x_n.x M_1 \dots M_k$  and  $\lambda x_1 \dots \lambda x_n.x M'_1 \dots M'_k$  can be said to match if all the pairs of “components”  $M_i$  and  $M'_i$  are related.

As an example, it is easy to prove that the call-by-value variants of Curry’s and Turing’s fixed-point combinators are eager normal form bisimilar (see Section 3).

**Eager normal form bisimulation up to  $\eta$ .** The definition of eager normal form bisimulation outlined in the previous section indeed describes what is called eager normal form bisimulation in Lassen [30]. It is a sound, but not complete, proof principle with respect to contextual equivalence. Analogously with the situation for head normal form bisimulation, a counterexample to completeness is given by the  $\eta_v$ -axiom: with the definition sketched above,  $x$  and  $\lambda y.yx$  are not bisimilar.

Accordingly, the definition of an eager normal form bisimulation that we will use in Section 3 is an “extensional” variant of the above definition; it defines

what is called an eager normal form bisimulation *up to  $\eta$*  in Lassen [30]. Informally, what changes is the definition of which values match: two values  $v$  and  $v'$  should match if the terms  $v x$  and  $v' x$  are related for some fresh variable  $x$ . To avoid introducing unnecessary redexes of the form  $(\lambda x.t) x$ , we use the following formulation:

- Values  $v$  and  $v'$  match if either:
  1.  $v$  and  $v'$  are the same variable.
  2.  $v = \lambda x.t$  and  $v' = \lambda x.t'$  where  $t$  and  $t'$  are related.
  3.  $v = y$  and  $v' = \lambda x.t'$  where  $y x$  and  $t'$  are related.
  4.  $v = \lambda x.t$  and  $v' = y$  where  $t$  and  $y x$  are related.

At the same time, it is convenient to reformulate the definition of which evaluation contexts match, in order to avoid introducing unnecessary redexes of the form  $\text{let } z=z \text{ in } t$ :

- Evaluation contexts  $E$  and  $E'$  match if either:
  1.  $E = E' = []$
  2.  $E = (\text{let } z=[] \text{ in } t)$  and  $E' = (\text{let } z=[] \text{ in } t')$  where  $t$  and  $t'$  are related.
  3.  $E = []$  and  $E' = (\text{let } z=[] \text{ in } t')$  where  $z$  and  $t'$  are related.
  4.  $E = (\text{let } z=[] \text{ in } t)$  and  $E' = []$  where  $t$  and  $z$  are related.

In Section 3 some notation,  $v \star x$  and  $E \star z$ , is introduced in order to avoid enumerating all the different cases above.

**Incompleteness.** In the previous section we sketched an extensional variant of eager normal form bisimulation that is defined formally in Section 3. Unlike the situation for head normal form bisimulation, however, this extensional variant is not complete with respect to contextual equivalence.

Let  $\Omega$  be an arbitrary diverging term, and let  $v$  be an arbitrary closed value. Consider the following pairs of terms [30]:

- $\Omega$  and  $\text{let } x=y v \text{ in } \Omega$
- $y v$  and  $\text{let } x=y v \text{ in } y v$

Neither of these two pairs of terms are eager normal form bisimilar. Informally, the reason is that eager normal form bisimilar terms must interact with their environment completely in lockstep; they must call the same unknown functions (here  $y$ ) the same number of times and in the same order.

On the other hand, both of the two pairs of terms are contextually equivalent. Informally, for the first pair of terms, the reason is that it does not matter what the function  $y$  does when applied to  $v$ . If it diverges, then the whole term diverges; and if it returns a result, then this result will be thrown away and the whole term will diverge anyway. For the second pair of terms, the reason is similar: if  $y v$  diverges, then both terms diverge; and if  $y v$  returns a result, then both terms return this result, only the term on the right computes it twice.

Both of these arguments can be made formal by using the operational extensibility property of the call-by-value  $\lambda$ -calculus [13, 46]: two open terms  $t$  and  $t'$  are contextually equivalent if and only if  $t[v_0/y]$  and  $t'[v_0/y]$  are contextually equivalent for every closed value  $v_0$ .

We return to the issue of incompleteness in Section 11.

**CPS correspondence.** As shown by Lassen [30], Plotkin’s call-by-value CPS transformation [49] yields a precise correspondence between eager normal form bisimulation and head normal form bisimulation (Böhm tree equivalence). Two terms of the call-by-value  $\lambda$ -calculus are eager normal form bisimilar in the “non-extensional” sense described in Section 2.3 if and only if their CPS transforms are head normal form bisimilar (have the same Böhm tree). Two terms are eager normal form bisimilar in the “extensional” sense described in Section 2.3 if and only if their CPS transforms are extensional head normal form bisimilar (have the same Böhm tree up to possibly infinite  $\eta$ -expansion).

Earlier, Boudol [9] showed that Plotkin’s call-by-name CPS transformation yields a similar correspondence between Lévy–Longo tree equivalence and Böhm tree equivalence.

**Adding control and state.** After having reviewed earlier work on normal form bisimulation and eager normal form bisimulation, we now briefly discuss the extension of eager normal form bisimulation to a calculus with control and state, as developed in the rest of this article.

In Section 2.3 we saw that eager normal form bisimulation is incomplete with respect to contextual equivalence. There are potentially two ways to overcome this problem: one is to make more terms bisimilar, the other is to make fewer terms contextually complete. In the rest of this article we consider the second approach: adding effects to the call-by-value  $\lambda$ -calculus not only brings us a bit closer to programming languages such as Scheme [22] or Standard ML [41], it also makes fewer pure terms contextually equivalent. Recall the two pairs of terms from Section 2.3:

- Adding control effects breaks the contextual equivalence of the first pair of terms. Intuitively, if the function  $y$  can, e.g., throw an exception, then the term on the right can escape divergence.
- Adding state effects breaks the contextual equivalence of the second pair of terms. Intuitively, the presence of mutable state allows the function  $y$  to return different results each time it is called.

As already mentioned, the extension of eager normal form bisimulation to a calculus with control effects and state effects that we present in this article is indeed both sound and complete. Completeness means that even though eager normal form bisimilarity is a very intensional form of equivalence, in the sense that bisimilar terms must interact with their environment completely in lockstep, it identifies as many terms as possible in our calculus with control and state effects. Furthermore, even though completeness hinges on the presence of an “exotic” control operator similar to the call/cc of Scheme and Standard ML of New Jersey<sup>5</sup>, almost every programming language will break contextual equivalences of the kinds considered in Section 2.3.

---

<sup>5</sup> <http://www.smlnj.org>

### 3 Eager Normal Form Bisimulation

We now turn to the main technical contributions of this article. In this section we give a formal definition of eager normal form (enf) bisimulation, and in the next sections we extend that definition to calculi with control and/or state effects.

Let us briefly reintroduce the definition of enf bisimulation for the pure call-by-value  $\lambda$ -calculus [30].

VARIABLES  $x, y, z$

VALUES  $v ::= x \mid \lambda x. t$

TERMS  $t ::= v \mid \text{let } x=t_1 \text{ in } t_2 \mid v_1 v_2$

We identify terms up to renaming of bound variables. Reduction is defined by means of evaluation contexts:

EVALUATION CONTEXTS  $E ::= [] \mid E[\text{let } x=[:] \text{ in } t]$

EAGER NORMAL FORMS (ENFS)  $e ::= v \mid E[x v]$

$$(R1) \quad E[\text{let } x=v \text{ in } t] \mapsto E[t[v/x]]$$

$$(R2) \quad E[(\lambda x. t) v] \mapsto E[t[v/x]]$$

The reflexive-transitive closure of the reduction relation  $\mapsto$  is written  $\mapsto^*$ . For every term  $t$ , there are two possibilities: either  $t$  *diverges* in the sense that there is an infinite reduction sequence starting from  $t$ , or else  $t$  *converges* in the sense that  $t \mapsto^* e$  for some (unique) eager normal form  $e$ . The notation  $t \mapsto^\omega$  means that  $t$  diverges. Eager normal forms are truly normal forms with respect to reduction: they do not reduce to anything.

For a syntactic phrase  $\phi$ , let  $\text{FV}(\phi)$  denote the set of free variables of  $\phi$  (the formal definitions are omitted).

**Definition 3.1.** A binary relation  $S$  on terms is an enf bisimulation if  $S \subseteq B(S)$ , where

$$B(S) = \{(t, t') \mid \text{either } t \mapsto^\omega \text{ and } t' \mapsto^\omega, \\ \text{or } t \mapsto^* e \text{ and } t' \mapsto^* e' \text{ where } (e, e') \in M(S)\}$$

$$\begin{aligned} M(S) = \{(v, v') \mid (v, v') \in V(S)\} \\ \cup \{(E[x v], E'[x v']) \mid (E, E') \in K(S) \& \\ (v, v') \in V(S)\} \end{aligned}$$

$$V(S) = \{(x, x)\} \cup \{(v, v') \mid \exists y \notin \text{FV}(v) \cup \text{FV}(v'). \\ (v \star y, v' \star y) \in S\}$$

$$K(S) = \{([], [])\} \cup \{(E, E') \mid \exists y \notin \text{FV}(E) \cup \text{FV}(E'). \\ (E \star y, E' \star y) \in S\}$$

where in turn

$$x \star y = x y$$

$$(\lambda y. t) \star x = t[x/y]$$

$$[] \star y = y$$

$$E[\text{let } y=[:] \text{ in } t] \star x = E[t[x/y]].$$

As pointed out on page 342 we use  $v \star y$  instead of  $v y$  and  $E \star y$  instead of  $E[y]$  in the definitions of  $V(S)$  and  $K(S)$  in order to avoid introducing unnecessary trivial redexes.

The intuition behind enf bisimulation is that two related open terms either (1) both diverge, or (2) reduce to matching eager normal forms whose components are again related. As an example, define the Curry call-by-value fixed-point combinator  $\Upsilon_v$ :

$$\begin{aligned}\Psi[f] &= \lambda g. f (\lambda x. \text{let } z=g \text{ in } z x) \\ \Upsilon_v &= \lambda f. \Psi[f] \Psi[f]\end{aligned}$$

and the Turing call-by-value fixed-point combinator  $\Theta_v$ :

$$\begin{aligned}\Xi &= \lambda g. \lambda f. f (\lambda x. \text{let } z_1=g \text{ in let } z_2=z_1 f \text{ in } z_2 x) \\ \Theta_v &= \Xi \Xi.\end{aligned}$$

These two fixed-point combinators are enf bisimilar, i.e., there exists an enf bisimulation  $S$  such that  $(\Upsilon_v, \Theta_v) \in S$  [30]. Indeed, it is easy to construct such an  $S$  by starting with the singleton  $\{(\Upsilon_v, \Theta_v)\}$  and then iteratively adding pairs in order to satisfy the definition of an enf bisimulation above. (In Section 6, a similar, but more complicated, equivalence between  $\Upsilon_v$  and a store-based fixed-point combinator is shown.)

**Remark.** The following construction, derived from the Turing call-by-value fixed-point combinator, is convenient for defining functions by recursion: For all values  $v$ ,  $v_1$ , and  $v_2$ , define

$$\begin{aligned}D[v_1, v_2] &= \text{let } z_1=\Theta_v \text{ in let } z_2=z_1 v_1 \text{ in } z_2 v_2 \\ \text{fix}[v] &= \lambda x. D[v, x]\end{aligned}$$

Then  $\text{fix}[v] x \mapsto^* \text{let } z=v \text{ fix}[v] \text{ in } z x$ . We will use these definitions of  $\text{fix}[v]$  and  $D[v_1, v_2]$  in the examples below.

Contextual equivalence is defined in the standard way. Informally, two terms  $t$  and  $t'$  are contextually equivalent if for every many-holed term context  $C[]$  such that  $C[t]$  and  $C[t']$  are closed terms,  $C[t]$  converges if and only if  $C[t']$  converges.

**Theorem 3.2 ([30]).** *If  $(t, t') \in S$  for some enf bisimulation  $S$ , then  $t$  and  $t'$  are contextually equivalent.*

(Recall that we call an enf bisimulation here is called an enf bisimulation up to  $\eta$  in Lassen [30].)

In the sequel we often omit the “enf” qualifier for bisimulations and instead qualify them by calculi. We will refer to the bisimulations for the pure call-by-value  $\lambda$ -calculus in Definition 3.1 as “ $\lambda$ -bisimulations”.

## 4 The $\lambda\mu$ -Calculus

We now extend enf bisimulation to the  $\lambda\mu$ -calculus. This extension is new, but based on head normal form bisimulation for the  $\lambda\mu$ -calculus [32]. Here we consider call-by-value reduction and not head reduction.

```
VARIABLES  $x, y, z$ 
NAMES  $a, b$ 
VALUES  $v ::= x \mid \lambda x. t$ 
NAMED TERMS  $nt ::= [a]t$ 
TERMS  $t ::= v \mid \text{let } x=t_1 \text{ in } t_2 \mid v_1 v_2 \mid \mu a. nt$ 
```

We identify syntactic phrases up to renaming of bound variables and names. For a syntactic phrase  $\phi$ , let  $\text{FN}(\phi)$  denote the set of free names of  $\phi$ .

Names in the  $\lambda\mu$ -calculus represent continuations. Names are not first-class, but we will represent a name  $a$  as the first-class value  $\hat{a} = \lambda x. \mu b. [a]x$ . The familiar *call/cc* control operator can be encoded in the  $\lambda\mu$ -calculus as

$$\text{call/cc} = \lambda f. \mu a. [a]f \hat{a}.$$

The operational semantics of the  $\lambda\mu$ -calculus is defined by a reduction relation on named terms:

```
NAMED EVAL. CONTEXTS  $NE ::= [a][] \mid NE[\text{let } x=[] \text{ in } t]$ 
NAMED ENFS  $ne ::= [a]v \mid NE[x v]$ 
```

- (R $\mu$ 1)  $NE[\text{let } x=v \text{ in } t] \mapsto NE[t[v/x]]$
- (R $\mu$ 2)  $NE[(\lambda x. t) v] \mapsto NE[t[v/x]]$
- (R $\mu$ 3)  $NE[\mu a. nt] \mapsto nt[NE/a]$

Here  $\phi[NE/a]$  denotes capture-avoiding substitution of named evaluation contexts for names: for example, if  $b \notin \text{FN}(NE)$ , then  $(\mu b. [a]t)[NE/a] = \mu b. NE[t]$ .

**Definition 4.1.** A binary relation  $S$  on named  $\lambda\mu$ -terms is a  $\lambda\mu$ -bisimulation if  $S \subseteq B_\mu(S)$ , where

$$B_\mu(S) = \{(nt, nt') \mid \text{either } nt \mapsto^\omega \text{ and } nt' \mapsto^\omega, \\ \text{or } nt \mapsto^* ne \text{ and } nt' \mapsto^* ne' \\ \text{where } (ne, ne') \in M_\mu(S)\}$$

$$M_\mu(S) = \{([a]v, [a]v') \mid (v, v') \in V_\mu(S)\} \\ \cup \{(NE[x v], NE'[x v']) \mid (NE, NE') \in K_\mu(S) \& \\ (v, v') \in V_\mu(S)\}$$

$$V_\mu(S) = \{(x, x)\} \\ \cup \{(v, v') \mid \exists y \notin \text{FV}(v) \cup \text{FV}(v'). \\ (v \star y, v' \star y) \in T_\mu(S)\}$$

$$\begin{aligned}
K_\mu(S) &= \{([a][], [a][])\} \\
&\cup \{(NE, NE') \mid \exists y \notin \text{FV}(NE) \cup \text{FV}(NE'). \\
&\quad (NE \star y, NE' \star y) \in T_\mu(S)\} \\
T_\mu(S) &= \{(t, t') \mid \exists a \notin \text{FN}(t) \cup \text{FN}(t'). \\
&\quad ([a]t, [a]t') \in S\}
\end{aligned}$$

with  $[a][] \star y = [a]y$  and  $NE[\text{let } x = [] \text{ in } t] \star y = NE[t[y/x]]$ .

Here  $NE \star y$  is analogous to  $E \star y$ : this notation is used in order to avoid unnecessary redexes of the form  $NE[\text{let } x = y \text{ in } t]$ .

**Definition 4.2.** Say that  $t$  and  $t'$  are  $\lambda\mu$ -bisimilar, written  $t \sim_\mu t'$ , if there exists a  $\lambda\mu$ -bisimulation  $S$  such that  $(t, t') \in T_\mu(S)$ .

We show in Section 11 that  $\lambda\mu$ -bisimilar terms are contextually equivalent.

Recall that  $\hat{a} = \lambda x. \mu b. [a]x$ . To illustrate  $\lambda\mu$ -bisimilarity we define the term  $\psi = \text{fix}[\mathsf{P}]$ , where

$$\mathsf{P} = \lambda f. \lambda x. \mu a. [a] \text{let } y = x \hat{a} \text{ in } f y.$$

The term  $\psi$  takes a function  $x$  as argument and applies  $x$  to successive arguments

$$x \hat{a}_1 \hat{a}_2 \dots$$

until  $x$  applies one of the  $\hat{a}_i$  to an argument  $v$ , in which case  $v$  is returned as the result of  $\psi x$ . On the other hand,  $\psi x$  diverges if  $x$  never applies any of its arguments, e.g., if  $x = \lambda y. \Omega$  or  $x = \text{fix}[\lambda f. \lambda y. f]$ .

**Remark.** A term with the behavior of  $\psi$  cannot be expressed in the pure call-by-value  $\lambda$ -calculus. To see this, consider the two functions

$$v = \lambda y. \text{let } z = y \text{ in } \Omega \quad \text{and} \quad v' = \lambda y. \Omega.$$

where  $\Omega = (\lambda x. x x)(\lambda x. x x)$ . They are contextually equivalent in the pure call-by-value  $\lambda$ -calculus. (This can be established using the operational extensibility property of the pure call-by-value  $\lambda$ -calculus [13, 46], because the term  $\text{let } z = v_0 \text{ in } \Omega$  diverges if  $v_0$  is any closed pure value.) But  $\psi$  can tell them apart:  $\psi v$  converges while  $\psi v'$  diverges.

A potential optimization of  $\psi$  is the following variant  $\psi'$  which returns straight to its final ‘‘return address’’ when  $x$  applies an argument (rather than returning from all the recursive invocations of the recursive function):  $\psi' = \lambda x. \mu a. [a] \text{fix}[\mathsf{P}'] x$ , where

$$\mathsf{P}' = \lambda f. \lambda x. \text{let } y = x \hat{a} \text{ in } f y$$

(Recall the definitions of  $\text{fix}[v]$  and  $\mathsf{D}[v_1, v_2]$  on page 345.) The optimization is correct up to enf bisimilarity, that is,  $\psi \sim_\mu \psi'$ , because

$$\begin{aligned}
S = \{ &([a]\psi, [a]\psi'), ([a]\mathsf{D}[\mathsf{P}, x], [a]\mu a. [a]\text{fix}[\mathsf{P}'] x), \\
&([b]\mu b. [a]x, \mu b. [a]x), ([a]\text{fix}[\mathsf{P}] y, [a]\text{fix}[\mathsf{P}'] y) \}
\end{aligned}$$

is a  $\lambda\mu$ -bisimulation.

## 5 The $\lambda\rho$ -Calculus

The  $\lambda\rho$ -calculus is obtained from the pure call-by-value  $\lambda$ -calculus by adding constructs for allocating a number of new reference cells, for storing a value in a reference cell, and for fetching the value from a reference cell.

```
VARIABLES  $x, y, z$ 
REFERENCES  $i, j$ 
VALUES  $v ::= x \mid \lambda x. t$ 
TERMS  $t ::= v \mid \text{let } x=t_1 \text{ in } t_2 \mid v_1 v_2 \mid \rho s. t \mid i:=v; t \mid !i$ 
STORES  $s ::= \{i_1:=v_1, \dots, i_n:=v_n\}$  ( $i_1, \dots, i_n$  are distinct)
```

Stores are identified up to reordering, and therefore a store can be considered as a finite map from references to values. Terms are identified up to renaming of bound variables and references: in the term  $\rho s. t$ , the references in the domain of  $s$  are considered bound in the range of  $s$  and in  $t$ . For a syntactic phrase  $\phi$ , let  $\text{FR}(\phi)$  be the set of references occurring free in  $\phi$ . A syntactic phrase is *reference-closed* if it contains no free references. Write  $\text{dom}(s)$  for the domain of the store  $s$ . If  $s$  and  $s'$  have disjoint domains,  $s \cdot s'$  denotes their disjoint union. If  $s = \{i:=v\} \cdot s'$ , let  $s(i) = v$  and  $s[i:=v'] = \{i:=v'\} \cdot s'$ .

Reduction is defined on *configurations*, which are pairs  $(s, t)$  of stores and terms such that  $\text{FR}(t) \subseteq \text{dom}(s)$ . (Configurations are not identified up to renaming of the domains of the stores, hence a configuration  $(s, t)$  should not be thought of as a term  $\rho s. t$ .)

```
EVALUATION CONTEXTS  $E ::= [] \mid E[\text{let } x=[] \text{ in } t]$ 
EAGER NORMAL FORMS (ENFS)  $e ::= v \mid E[x v]$ 
```

- (R $\rho$ 1)  $(s, E[\text{let } x=v \text{ in } t]) \mapsto (s, E[t[v/x]])$
- (R $\rho$ 2)  $(s, E[(\lambda x. t) v]) \mapsto (s, E[t[v/x]])$
- (R $\rho$ 3)  $(s, E[\rho s'. t]) \mapsto (s \cdot s', E[t]),$   
if  $(\text{dom}(s) \cup \text{FR}(s) \cup \text{FR}(E)) \cap \text{dom}(s') = \emptyset$
- (R $\rho$ 4)  $(s, E[i:=v; t]) \mapsto (s[i:=v], E[t]) \quad \text{if } i \in \text{dom}(s)$
- (R $\rho$ 5)  $(s, E[!i]) \mapsto (s, E[s(i)]) \quad \text{if } i \in \text{dom}(s)$

Eager normal form bisimulation for the  $\lambda\rho$ -calculus is based on the relation-sets bisimulation approach [25, 29, 61]. Briefly, instead of defining a bisimulation as a single binary relation on terms, one defines a bisimulation as a *set* of such relations, each associated with a “world”: here, a pair of stores. The requirement is that if two terms are related in a certain world, then the eager normal forms (if any) of these two terms are related in a “future world” where the two stores may have changed. Moreover, a monotonicity condition must be satisfied: everything that was related in the old world must still be related in the new world.

Now for the formal definitions. Let  $X, Y, Z$  range over finite sets of variables and let  $J$  range over finite sets of references. We write  $X \cdot Y$  for the disjoint union of  $X$  and  $Y$ . When the meaning is clear from the context, we write a singleton set  $\{x\}$  as just  $x$ . We use the same notational conventions for finite sets of references.

Notation  $X, J \vdash \phi, \phi', \dots$  means the syntactic phrases  $\phi, \phi', \dots$  have free variables in  $X$  and free references in  $J$ . We omit  $X$  and/or  $J$  on the left of  $\vdash$  if it is empty.

Let  $R$  range over sets of triples  $(X|t, t')$ , more specifically over subsets of  $Rel(Y, J, J')$  for some  $Y, J$  and  $J'$ , where

$$\begin{aligned} Rel(Y, J, J') = \\ \{(X|t, t') \mid X \cap Y = \emptyset \text{ & } X \cdot Y, J \vdash t \text{ & } X \cdot Y, J' \vdash t'\} \end{aligned}$$

We identify triples that differ only up to renaming of the variables from the first component  $X$ : in the triple  $(X|t, t')$ , the variables in  $X$  are considered bound in  $t$  and  $t'$ . A triple  $(\emptyset|t, t')$  where the first component is empty is also written  $(|t, t')$ .

A *term relation tuple* is a quadruple  $(X|s, s', R)$  where  $X \vdash s, s'$  and where  $R \subseteq Rel(X, dom(s), dom(s'))$ . We identify term relation tuples that differ only up to renaming of the variables from the first component  $X$  and up to renaming of references. Let  $Q$  range over *term relation sets*, that is, sets of term relation tuples.

**Definition 5.1.**  $Q$  is a  $\lambda\rho$ -bisimulation iff  $Q \subseteq B_\rho(Q)$ , where

$$\begin{aligned} B_\rho(Q) = \{ & (X|s_0, s'_0, R_0) \mid \\ & \text{for all } (Y|t, t') \in R_0, \text{ either} \\ & (s_0, t) \mapsto^\omega \& (s'_0, t') \mapsto^\omega, \text{ or} \\ & \exists s_1, s'_1, e, e', R_1 \supseteq R_0, X_1 \supseteq X \cdot Y. \\ & (s_0, t) \mapsto^* (s_1, e) \& (s'_0, t') \mapsto^* (s'_1, e') \& \\ & (e, e') \in M_\rho(R_1) \& (X_1|s_1, s'_1, R_1) \in Q \} \end{aligned}$$

$$M_\rho(R) = \{ (v, v'), (E[x v], E'[x v']) \mid \\ (v, v') \in V_\rho(R) \& (E, E') \in K_\rho(R) \}$$

$$\begin{aligned} V_\rho(R) = \{ & (x, x) \} \\ & \cup \{ (v, v') \mid \exists y \notin FV(v) \cup FV(v'). \\ & (y|v \star y, v' \star y) \in R \} \end{aligned}$$

$$\begin{aligned} K_\rho(R) = \{ & ([], []) \} \\ & \cup \{ (E, E') \mid \exists y \notin FV(E) \cup FV(E'). \\ & (y|E \star y, E' \star y) \in R \} \end{aligned}$$

**Definition 5.2.** Reference-closed  $\lambda\rho$ -terms  $t$  and  $t'$  are  $\lambda\rho$ -bisimilar, written  $t \sim_\rho t'$ , iff there exists a  $\lambda\rho$ -bisimulation  $Q$  which contains a quadruple  $(X|\{\}, \{\}, R)$  with  $(|t, t') \in R$ .

We show in Section 10 that  $\lambda\rho$ -bisimilarity is a congruence. Therefore, as explained in Section 11,  $\lambda\rho$ -bisimilar terms are contextually equivalent.

Here is an example illustrating the monotonicity condition  $R_1 \supseteq R_0$  in the definition of  $B_\rho$ . Let  $I = \lambda x.x$  be the identity combinator, and let  $\Omega$  be an

arbitrary closed, diverging term. Abbreviate  $(!i) y = (\text{let } y_0 = !i \text{ in } y_0 y)$ . Consider the following two terms:

$$\begin{aligned} t &= \rho\{\iota:=I\}. \text{let } z=x (\lambda y. (!i) y) \text{ in } (\iota:=\lambda z. \Omega; x_0 I) \\ t' &= \text{let } z=x (\lambda y. I y) \text{ in } x_0 I \end{aligned}$$

We now sketch a proof that these two terms are not  $\lambda\rho$ -bisimilar. First, reduce  $t$  and  $t'$  together with the empty store:

$$\begin{aligned} (\{\}, t) &\mapsto^* (\{\iota:=I\}, \text{let } z=x (\lambda y. (!i) y) \text{ in } (\iota:=\lambda z. \Omega; x_0 I)) \\ (\{\}, t') &\mapsto^* (\{\}, \text{let } z=x (\lambda y. I y) \text{ in } x_0 I) \end{aligned}$$

We have reached two eager normal forms  $E[x (\lambda y. (!i) y)]$  and  $E'[x (\lambda y. I y)]$  where  $E = (\text{let } z=[] \text{ in } (\iota:=\lambda z. \Omega; x_0 I))$  and  $E' = (\text{let } z=[] \text{ in } x_0 I)$ . Both the pair of values  $\lambda y. (!i) y$  and  $\lambda y. I y$  and the pair of evaluation contexts  $E$  and  $E'$  must match. Let us consider the evaluation contexts first:

$$\begin{aligned} (\{\iota:=I\}, (\iota:=\lambda z. \Omega; x_0 I)) &\mapsto^* (\{\iota:=\lambda z. \Omega\}, x_0 I) \\ (\{\}, x_0 I) &\mapsto^* (\{\}, x_0 I) \end{aligned}$$

We have reached two identical eager normal forms  $x_0 I$  and  $x_0 I$ ; matching them is unproblematic. However, the monotonicity condition implies that the two values  $\lambda y. (!i) y$  and  $\lambda y. I y$  that we saw earlier must still match. This is impossible, as the configuration  $(\{\iota:=\lambda z. \Omega\}, (!i) y)$  diverges, while the configuration  $(\{\}, I y)$  converges.

We return to this example when discussing completeness in Section 11.

## 6 Example: Imperative Fixed-Point Combinator

It is well-known that a store that may contain functional values can be used to define functions by recursion. Abbreviate

$$\Pi[f, i] = \lambda x. \text{let } z_1 = !i \text{ in let } z_2 = f z_1 \text{ in } z_2 x$$

and consider the term:

$$Y_\rho = \lambda f. \rho\{\iota:=\Pi[f, i]\}. f \Pi[f, i].$$

$Y_\rho$  can be used to define functions by recursion in the  $\lambda\rho$ -calculus. The technique of defining recursive functions by means of a “circular store” is due to Landin [28].

We now show that the fixed-point combinator  $Y_\rho$  is  $\lambda\rho$ -bisimilar to the Curry call-by-value fixed-point combinator  $Y_v$  (defined in Section 3 above). This equivalence can be shown directly from the definition of a  $\lambda\rho$ -bisimulation, but it is more convenient to apply the following general lemma:

**Lemma 6.1.** *Define  $\hat{\rho}s. t = \rho s. t$  for  $s \neq \{\}$ , and  $\hat{\rho}\{\}. t = t$ . Assume that there exists a  $\lambda\rho$ -bisimulation containing a tuple  $(X|s, s', R)$  where  $(|t, t') \in R$ , and let  $x_1, \dots, x_n \in X$ . Then  $\lambda x_1 \dots \lambda x_n. \hat{\rho}s. t \sim_\rho \lambda x_1 \dots \lambda x_n. \hat{\rho}s'. t'$ .*

The lemma follows from Corollary 10.20 in Section 10.

**Proposition 6.2.**  $\mathbf{Y}_\rho \approx_\rho \mathbf{Y}_v$ .

*Proof.* By definition,  $\mathbf{Y}_\rho = \lambda f. \rho\{\iota := \Pi[f, \iota]\}. f \Pi[f, \iota]$  and  $\mathbf{Y}_v = \lambda f. \Psi[f] \Psi[f]$ . The proof therefore consists of constructing a  $\lambda\rho$ -bisimulation  $Q$  containing a tuple  $(\{f\}|\{\iota := \Pi[f, \iota]\}, \{\}, R)$  where  $(|f \Pi[f, \iota], \Psi[f] \Psi[f]) \in R$ , and then using Lemma 6.1.

Instead of specifying  $Q$  right away, we show how one would in practice construct  $Q$ : by starting from the two configurations  $(\{\iota := \Pi[f, \iota]\}, f \Pi[f, \iota])$  and  $(\{\}, \Psi[f] \Psi[f])$  and iteratively adding tuples in order to satisfy the conditions in the definition of a  $\lambda\rho$ -bisimulation. In that way, the main part of the equivalence proof consists in a number of calculations of reduction sequences.

Abbreviate  $D[f] = \lambda x. \text{let } z = \Psi[f] \Psi[f] \text{ in } z x$ . Now calculate:

$$\begin{aligned} (\{\iota := \Pi[f, \iota]\}, f \Pi[f, \iota]) &\mapsto^* (\{\iota := \Pi[f, \iota]\}, f D[f]) \\ (\{\}, \Psi[f] \Psi[f]) &\mapsto^* (\{\}, f D[f]). \end{aligned}$$

The two resulting eager normal forms are  $f \Pi[f, \iota]$  and  $f D[f]$ . The variables in function position match (both are  $f$ ), so consider the arguments,  $\Pi[f, \iota]$  and  $D[f]$ . Since

$$\Pi[f, \iota] = \lambda x. \text{let } z_1 = !\iota \text{ in let } z_2 = f z_1 \text{ in } z_2 x$$

and

$$D[f] = \lambda x. \text{let } z = \Psi[f] \Psi[f] \text{ in } z x,$$

the definition of a  $\lambda\rho$ -bisimulation indicates that one should continue by reducing the bodies of these two  $\lambda$ -abstractions:

$$\begin{aligned} (\{\iota := \Pi[f, \iota]\}, \text{let } z_1 = !\iota \text{ in let } z_2 = f z_1 \text{ in } z_2 x) \\ \mapsto^* (\{\iota := \Pi[f, \iota]\}, \text{let } z_2 = f \Pi[f, \iota] \text{ in } z_2 x) \end{aligned}$$

and

$$\begin{aligned} (\{\}, \text{let } z = \Psi[f] \Psi[f] \text{ in } z x) &\mapsto^* (\{\}, \text{let } z = f D[f] \text{ in } z x) \\ &= (\{\}, \text{let } z_2 = f D[f] \text{ in } z_2 x) \end{aligned}$$

The resulting two eager normal forms are

$$\text{let } z_2 = f \Pi[f, \iota] \text{ in } z_2 x \quad \text{and} \quad \text{let } z_2 = f D[f] \text{ in } z_2 x.$$

Again, the variables in function position match (both are  $f$ ), and the evaluation contexts are identical (both are  $\text{let } z_2 = [] \text{ in } z_2 x$ ). The function arguments,  $\Pi[f, \iota]$  and  $D[f]$ , are  $\lambda$ -abstractions, and therefore one should continue reducing the bodies of these two  $\lambda$ -abstractions. But this is exactly what was already done in the previous two reduction sequences.

Using the results of these calculations it is possible to construct the required bisimulation  $Q$ . First, define

$$\begin{aligned} R = \{ & (|f \Pi[f, i], \Psi[f] \Psi[f]), \\ & (x | \text{let } z_1 = !i \text{ in let } z_2 = f z_1 \text{ in } z_2 x, \\ & \quad \text{let } z = \Psi[f] \Psi[f] \text{ in } z x) \}. \end{aligned}$$

Let  $x_1, x_2, \dots$  be distinct variables, and define, for every  $n \geq 0$ ,

$$S_n = \{(z_2 | z_2 x_k, z_2 x_k) \mid 1 \leq k \leq n\}.$$

Finally, define  $Q$  as the set of all tuples

$$(\{f, x_1, \dots, x_n\} | \{i := \Pi[f, i]\}, \{\}, R \cup S_n)$$

where  $n \geq 0$ . Then  $Q$  is a  $\lambda\rho$ -bisimulation, as can be verified using the calculations above.

Note that  $Q$  contains the tuple  $(\{f\} | \{i := \Pi[f, i]\}, \{\}, R)$  where

$$(|f \Pi[f, i], \Psi[f] \Psi[f]) \in R.$$

Therefore, Lemma 6.1 implies that  $\Upsilon_\rho \approx_\rho \Upsilon_v$ . □

## 7 The $\lambda\mu\rho$ -Calculus

The  $\lambda\mu\rho$ -calculus combines the control aspects of the  $\lambda\mu$ -calculus with the state aspects of the  $\lambda\rho$ -calculus. The definition of  $\lambda\mu\rho$ -bisimilarity is a natural combination of the definitions of  $\lambda\mu$ -bisimilarity and of  $\lambda\rho$ -bisimilarity.

```

VARIABLES x, y, z
NAMES a, b
REFERENCES i, j
VALUES v ::= x | λx. t
NAMED TERMS nt ::= [a]t
TERMS t ::= v | let x=t1 in t2 | v1 v2 | μa. nt |
          ρs. t | i:=v; t | !i
STORES s ::= {i1=v1, ..., in=vn}

```

Reduction is defined on *configurations*, which are now pairs  $(s, nt)$  of stores and named terms such that  $\text{FR}(nt) \subseteq \text{dom}(s)$ .

```

NAMED EVAL. CONTEXTS NE ::= [a][] | NE[let x=[] in t]
NAMED ENFS ne ::= [a]v | NE[x v]

```

- (Rμρ1)  $(s, NE[\text{let } x=v \text{ in } t]) \mapsto (s, NE[t[v/x]])$
- (Rμρ2)  $(s, NE[(\lambda x. t) v]) \mapsto (s, NE[t[v/x]])$
- (Rμρ3)  $(s, NE[\mu a. nt]) \mapsto (s, nt[NE/a])$

- (R $\mu\rho$ 4)  $(s, NE[\rho s'. t]) \mapsto (s \cdot s', NE[t]),$   
     if  $(dom(s) \cup FR(s) \cup FR(NE)) \cap dom(s') = \emptyset$
- (R $\mu\rho$ 5)  $(s, NE[i := v; t]) \mapsto (s[i := v], NE[t])$  if  $i \in dom(s)$
- (R $\mu\rho$ 6)  $(s, NE[!i]) \mapsto (s, NE[s(i)])$  if  $i \in dom(s)$

Now  $X, Y, Z$  range over finite sets of variables and names. Let  $NR$  range over sets of triples  $(X|nt, nt')$ , more specifically subsets of  $NRel(Y, J, J')$  for some  $Y$ ,  $J$  and  $J'$ , where

$$\begin{aligned} NRel(Y, J, J') = \\ \{(X|nt, nt') \mid X \cap Y = \emptyset \& X \cdot Y, J \vdash nt \& X \cdot Y, J' \vdash nt'\} \end{aligned}$$

We identify triples that differ only up to renaming of the variables and names from the first component  $X$ .

A *named term relation tuple* is a quadruple  $(X|s, s', NR)$  where  $X \vdash s, s'$  and  $NR \subseteq NRel(X, dom(s), dom(s'))$ . We identify named term relation tuples that differ only up to renaming of the variables and names from the first component  $X$  and up to renaming of references. A *named term relation set* is a set of named term relation tuples. Let  $NQ$  range over named term relations sets.

**Definition 7.1.**  $NQ$  is a  $\lambda\mu\rho$ -bisimulation iff  $NQ \subseteq B_{\mu\rho}(NQ)$ , where

$$\begin{aligned} B_{\mu\rho}(NQ) = \{(X|s_0, s'_0, NR_0) \mid \\ \text{for all } (Y|nt, nt') \in NR_0, \text{ either} \\ (s_0, nt) \mapsto^\omega \& (s'_0, nt') \mapsto^\omega, \text{ or} \\ \exists s_1, s'_1, ne, ne', NR_1 \supseteq NR_0, X_1 \supseteq X \cdot Y. \\ (s_0, nt) \mapsto^* (s_1, ne) \& \\ (s'_0, nt') \mapsto^* (s'_1, ne') \& \\ (ne, ne') \in M_{\mu\rho}(NR_1) \& \\ (X_1|s_1, s'_1, NR_1) \in NQ\} \end{aligned}$$

$$M_{\mu\rho}(NR) = \{([a]v, [a]v'), (NE[x v], NE'[x v']) \mid \\ (v, v') \in V_{\mu\rho}(NR) \& (NE, NE') \in K_{\mu\rho}(NR)\}$$

$$\begin{aligned} V_{\mu\rho}(NR) = \{(x, x)\} \\ \cup \{(v, v') \mid \exists y \notin FV(v) \cup FV(v'). \\ \exists a \notin FN(v) \cup FN(v'). \\ (a \cdot y | [a](v \star y), [a](v' \star y)) \in NR\} \end{aligned}$$

$$\begin{aligned} K_{\mu\rho}(NR) = \{([a][], [a][])\} \\ \cup \{(NE, NE') \mid \exists y \notin FV(NE) \cup FV(NE'). \\ (y | NE \star y, NE' \star y) \in NR\} \end{aligned}$$

**Definition 7.2.** Reference-closed named terms  $nt$  and  $nt'$  are  $\lambda\mu\rho$ -bisimilar, written  $nt \sim_{\mu\rho} nt'$ , iff there exists a  $\lambda\mu\rho$ -bisimulation  $NQ$  which contains a quadruple  $(X|\{\}, \{\}, NR)$  with  $(|nt, nt') \in NR$ . Reference-closed terms  $t$  and  $t'$  are  $\lambda\mu\rho$ -bisimilar, written  $t \sim_{\mu\rho} t'$ , iff there exists a  $\lambda\mu\rho$ -bisimulation  $NQ$  which contains a quadruple  $(X|\{\}, \{\}, NR)$  with  $(t, t') \in T_{\mu\rho}(NR)$ , where

$$T_{\mu\rho}(NR) = \{(t, t') \mid \exists a \notin FN(t) \cup FN(t'). (a | [a]t, [a]t') \in NR\}.$$

We show in Section 10 that  $\lambda\mu\rho$ -bisimilarity is a congruence and in Section 11 that it coincides with contextual equivalence.

## 8 Example: One-Shot Continuations

As an extended example, we show the correctness of Friedman and Haynes's encoding of call/cc in terms of “one-shot continuations” [16].

A one-shot continuation is a continuation which may be applied at most once. Friedman and Haynes showed that, perhaps surprisingly, call/cc can be encoded in terms of its restricted one-shot variant. They did this by exhibiting an “extraordinarily difficult program” [16, p.248] together with an informal equivalence argument. We confirm the correctness of this program by a formal proof using the enf bisimulation method. The equivalence proof below can be viewed as a formalization of Friedman and Haynes's informal argument.

One cannot directly use the  $\lambda\mu\rho$ -calculus to prove correctness of this encoding of call/cc, since the  $\lambda\mu\rho$ -calculus does not contain one-shot continuations as a primitive. Instead, we define one-shot continuations in terms of unrestricted continuations using another, but simpler, construction due to Friedman and Haynes. We then show the correctness of the encoding of call/cc by means of one-shot continuations relative to this encoding of one-shot continuations.

First, we need to encode a conditional operator in the  $\lambda\mu\rho$ -calculus. Since the evaluation order in the  $\lambda\mu\rho$ -calculus is call-by-value, the encoding is done using “thunks”:

$$\begin{aligned} T &= \lambda x. \lambda y. x \mathsf{I} \\ F &= \lambda x. \lambda y. y \mathsf{I} \end{aligned}$$

$$\begin{aligned} \text{if}[t_1, t_2, t_3] &= \text{let } z_1 = t_1 \text{ in} \\ &\quad \text{let } z_2 = z_1 (\lambda z. t_2) \text{ in} \\ &\quad z_2 (\lambda z. t_3) \end{aligned}$$

where  $\mathsf{I} = \lambda x. x$ , and where  $z_1$  and  $z_2$  are not free in  $t_1$ ,  $t_2$ , or  $t_3$ .

Recall the definition of call/cc:

$$\text{call/cc} = \lambda f. \mu a. [a]f \hat{a}$$

where  $\hat{a} = \lambda x. \mu b. [a]x$ . Now define the one-shot variant of call/cc:

$$\text{call/cc1} = \lambda f. (\text{call/cc} (\lambda k. \rho\{\iota:=T\}. f (\lambda x. \text{if}[\mathsf{!}\iota, (\iota:=F; k x), \Omega])))$$

The requirement that every captured continuation  $k$  is applied at most once is enforced by means of the local reference  $\iota$ .

Now for the encoding of unrestricted continuations by means of one-shot continuations. For every reference  $\jmath$ , define

$$\begin{aligned} \Phi_\jmath &= \lambda g. \lambda f. \text{let } y = \text{call/cc1} (\lambda k. (\jmath := k; f (\lambda x. \text{let } y = \mathsf{!}\jmath \text{ in } y x))) \\ &\quad \text{in } \text{call/cc1} (\lambda k'. g (\lambda k. k' y)). \end{aligned}$$

Then define

$$\text{call/cc*} = \lambda f. \rho\{\jmath:=\text{l}\}. \text{fix}[\Phi_\jmath] f.$$

(See the original presentation of the encoding [16] for an informal explanation of how it works.)

The aim of this section is to show that

$$\text{call/cc} \sim_{\mu\rho} \text{call/cc*}.$$

It follows that  $\text{call/cc}$  and  $\text{call/cc*}$  are contextually equivalent, and hence that  $\text{call/cc*}$  is as an encoding of  $\text{call/cc}$  by means of one-shot continuations.

As in Section 6, the equivalence could be shown directly from the definition of a bisimulation, but it is more convenient to use the following generalization of Lemma 6.1 to the  $\lambda\mu\rho$ -calculus:

**Lemma 8.1.** *Define  $\hat{\rho}s.t = \rho s.t$  for  $s \neq \{\}$ , and  $\hat{\rho}\{\}.t = t$ . Assume that there exists a  $\lambda\mu\rho$ -bisimulation containing a tuple  $(X|s, s', NR)$  where  $([a]t, [a]t') \in NR$ , and let  $x_1, \dots, x_n \in X$ . If  $a \in X$  does not occur free in any of  $s$ ,  $s'$ ,  $t$ , and  $t'$ , then  $\lambda x_1 \dots \lambda x_n. \hat{\rho}s.t \sim_{\mu\rho} \lambda x_1 \dots \lambda x_n. \hat{\rho}s'.t'$ .*

The lemma follows from Corollary 10.20 in Section 10.

**Proposition 8.2.**  $\text{call/cc} \sim_{\mu\rho} \text{call/cc*}$ .

*Proof.* By definition:

$$\begin{aligned} \text{call/cc} &= \lambda f. \mu a. [a]f \hat{a} \\ \text{call/cc*} &= \lambda f. \rho\{\jmath:=\text{l}\}. \text{fix}[\Phi_\jmath] f. \end{aligned}$$

We therefore construct a bisimulation containing a tuple

$$(f \cdot a | \{\}, \{\jmath:=\text{l}\}, NR)$$

where  $([a] \mu a. [a]f \hat{a}, [a] \text{fix}[\Phi_\jmath] f) \in NR$ . The conclusion then follows from the lemma above.

The main part of the proof consists in a number of calculations of reduction sequences. One starts from the two configurations  $(\{\}, [a] \mu a. [a]f \hat{a})$  and  $(\{\jmath:=\text{l}\}, [a] \text{fix}[\Phi_\jmath] f)$  and iteratively adds tuples in order to satisfy the conditions in the definition of a  $\lambda\mu\rho$ -bisimulation.

First, define the named evaluation context

$$NE_0 = [a] \text{let } x=[\ ] \text{ in call/cc1} (\lambda k'. \text{fix}[\Phi_\jmath] (\lambda k. k'x))$$

and for every reference  $\iota$ , define the term

$$C[\iota] = \lambda x. \text{if}[\iota, (\iota:=F; (\lambda x. \mu b. NE_0[x]) x), \Omega].$$

Now calculate, for any store  $s$  and any value  $v$ :

- (1)  $(s \cdot \{j:=v\}, [a]\text{fix}[\Phi_j] f)$   
 $\xrightarrow{*}$   
 $(s \cdot \{j:=C[i], v:=\top\}, NE_0[f (\lambda x. \text{let } y=!\_j \text{ in } y x)]).$
- (2)  $(s \cdot \{j:=C[i], v:=\top\}, [b] \text{let } y=!\_j \text{ in } y x)$   
 $\xrightarrow{*}$   
 $(s \cdot \{j:=C[i], v:=F\}, [a]\text{call/cc1}(\lambda k'. \text{fix}[\Phi_j](\lambda k. k' x))).$
- (3)  $(s \cdot \{j:=C[i]\}, [a]\text{call/cc1}(\lambda k'. \text{fix}[\Phi_j](\lambda k. k' x)))$   
 $\xrightarrow{*}$   
 $(s \cdot \{j:=C[i'], v_0:=F, i':=\top\}, [a]x).$

These calculations dictate the following construction of a  $\lambda\mu\rho$ -bisimulation: let

$$NR_0 = \{([a]\mu a. [a]f \hat{a}, [a]\text{fix}[\Phi_j] f), \\ (y \mid [a]y, [a]\text{call/cc1}(\lambda k'. \text{fix}[\Phi_j](\lambda k. k' y))), \\ (y \cdot b \mid [b]\mu b.[a]y, [b] \text{let } z=!\_j \text{ in } z y)\}$$

and let  $NQ$  consist of the tuple

$$(f \cdot a \mid \{\}, \{j:=\mathbb{I}\}, \{([a]\mu a. [a]f \hat{a}, [a]\text{fix}[\Phi_j] f)\})$$

together with all named term relation tuples of the form

$$(X \mid \{\}, s, NR_0)$$

where  $\{f, a\} \subseteq X$ , where  $s$  is a store such that  $j \in \text{dom}(s)$ , and where there exists an  $i \in \text{dom}(s)$  such that

$$s(j) = C[i] \quad \text{and} \quad s(i) = \top.$$

Then  $NQ$  is a  $\lambda\mu\rho$ -bisimulation, as can be verified using the calculations (1)-(3) above. By Lemma 8.1,  $\text{call/cc} \sim_{\mu\rho} \text{call/cc*}$ .  $\square$

## 9 Enf Bisimulation for Terms with Free References

So far in this article, eager normal form bisimulation has been used as a proof principle for proving equivalence of *reference-closed* terms. In this section we show how to extend eager normal form bisimulation to terms which may contain free references. Besides allowing one to prove equivalences about terms with free references, this extension is also used in the congruence proof for enf bisimilarity in Section 10. As a part of that proof, it must be shown that the following holds: If  $t \sim_{\mu\rho} t'$  and  $v \sim_{\mu\rho} v'$ , then  $\rho\{i:=v\}.t \sim_{\mu\rho} \rho\{i:=v'\}.t'$  and  $i:=v; t \sim_{\mu\rho} i:=v'; t'$ . Here the reference  $i$  will in general occur free in the terms  $t$ ,  $t'$ ,  $v$ , and  $v'$ , and, of course, in the terms  $i:=v; t$  and  $i:=v'; t'$ .

The modification needed to take free references into account can be explained as follows. Suppose that the free references of the terms  $t$  and  $t'$  are contained

in  $J$ , and that one wants to prove that  $t$  and  $t'$  are equivalent. According to the previous definition, one requirement is that  $[a]t$  and  $[a]t'$  should either both diverge, or reduce to matching named eager normal forms. But one cannot reduce  $[a]t$  and  $[a]t'$  without providing values for the references in  $J$ , i.e., the references which are free in  $t$  and  $t'$ . The solution is to initialize the references in  $J$  with a number of fresh variables  $z_j{}^{j \in J}$ . This initialization takes care of the “input” aspect of the free references; the “output” aspect is taken care of by an extra requirement: if both  $(\{j := z_j{}^{j \in J}\}, [a]t)$  and  $(\{j := z_j{}^{j \in J}\}, [a]t')$  reduce to named eager normal forms, then in the two resulting stores, the references from  $J$  must contain values which are pairwise related.

Now for the formal definitions. Named term relation sets are generalized as follows: let

$$\begin{aligned} NU_J = \{ & (X|s, s', NR) \mid \\ & X, J \vdash s, s' \text{ &} \\ & NR \subseteq NRel(X, J \cdot \text{dom}(s), J \cdot \text{dom}(s')) \}. \end{aligned}$$

We identify quadruples that differ only up to renaming of the variables and names from the first component  $X$  and up to renaming of references from  $\text{dom}(s)$  and  $\text{dom}(s')$ .

**Definition 9.1.**  $NQ \subseteq NU_J$  is a  $J$ -bisimulation iff  $NQ \subseteq B_J(NQ)$ , where

$$\begin{aligned} B_J(NQ) = \{ & (X|s_0, s'_0, NR_0) \in NU_J \mid \\ & \text{for all distinct variables } z_i{}^{i \in J} \\ & \text{and all } (Y|nt, nt') \in NR_0, \text{ either} \\ & (\{i := z_i{}^{i \in J}\} \cdot s_0, nt) \mapsto^\omega \& (\{i := z_i{}^{i \in J}\} \cdot s'_0, nt') \mapsto^\omega, \text{ or} \\ & \exists ne, ne', (v_i, v'_i){}^{i \in J}, s_1, s'_1, NR_1 \supseteq NR_0, X_1 \supseteq X \cdot Y \cdot z_i{}^{i \in J}. \\ & (\{i := z_i{}^{i \in J}\} \cdot s_0, nt) \mapsto^* (\{i := v_i{}^{i \in J}\} \cdot s_1, ne) \& \\ & (\{i := z_i{}^{i \in J}\} \cdot s'_0, nt') \mapsto^* (\{i := v'_i{}^{i \in J}\} \cdot s'_1, ne') \& \\ & (ne, ne') \in M_{\mu\rho}(NR_1) \& \\ & \forall i \in J. (v_i, v'_i) \in V_{\mu\rho}(NR_1) \& \\ & (X_1|s_1, s'_1, NR_1) \in NQ \} \end{aligned}$$

Say that two terms  $t$  and  $t'$  are  $J$ -bisimilar if there exists a  $J$ -bisimulation containing a tuple  $(X|\{\}, \{\}, NR)$  where  $(t, t') \in T_{\mu\rho}(NR)$ .

We now generalize the previously given definition of enf bisimilarity for reference-closed terms:

**Definition 9.2.** Let  $t$  and  $t'$  be  $\lambda\mu\rho$ -terms. Say that  $t$  and  $t'$  are  $\lambda\mu\rho$ -bisimilar, written  $t \sim_{\mu\rho} t'$ , if there exists a finite set  $J$  of references such that  $t$  and  $t'$  are  $J$ -bisimilar.

*Example 9.3.* It is easy to show that

$$\text{let } z = !j \text{ in } (j := l; j := z; f x) \sim_{\mu\rho} f x$$

while on the other hand

$$\text{let } z = !\gamma \text{ in } (\gamma := \text{I}; \text{let } y = f x \text{ in } (\gamma := z; y)) \not\sim_{\mu\rho} f x.$$

The proofs of this equivalence and this non-equivalence illustrate a basic sequentiality property of the calculi considered in this article: in order for two terms to be equivalent, it is enough that the contents of the free references are equivalent at certain “synchronization points,” but in-between these points the contents of the free references can be modified arbitrarily.

**Proposition 9.4.** *Let  $J_0$  and  $J$  be finite sets of references such that  $J_0 \subseteq J$ . Any two terms which are  $J_0$ -bisimilar are also  $J$ -bisimilar.*

## 10 Congruence

This section outlines the proof that  $\lambda\mu\rho$ -bisimilarity is a congruence: it is an equivalence relation which is furthermore compatible. A binary relation  $S$  on terms and named terms of the  $\lambda\mu\rho$ -calculus is *compatible* if it is closed under the term formation rules of the  $\lambda\mu\rho$ -calculus. For example, if  $t_1 S t'_1$  and  $t_2 S t'_2$ , then also  $(\text{let } x = t_1 \text{ in } t_2) S (\text{let } x = t'_1 \text{ in } t'_2)$ , and if  $nt S nt'$ , then  $\mu a. nt S \mu a. nt'$ . The straightforward formal definition is omitted.

### 10.1 Overview and Related Proofs

Let us begin with a brief overview of our congruence proof and how it relates to similar proofs in the literature.

First of all, we cannot adapt Howe’s method for proving congruence of applicative bisimilarity [19] to normal form bisimilarity, because the proof of substitutivity for Howe’s congruence candidate relation breaks down. It requires that bisimilarity is closed under “closed instantiations”. This holds for applicative bisimilarity by its definition, namely by its universal quantification over closed arguments and substitutions, and it does not hold for normal form bisimilarity.

Our congruence proof is based on a proof for eager normal form bisimilarity in the pure call-by-value  $\lambda$ -calculus [30], a proof which is in turn based on an earlier congruence proof [34] for head normal form bisimilarity (recall Section 2.2). However, those earlier proofs do not immediately generalize to the  $\lambda\rho$ -calculus because of the stateful features in that calculus.

It turns out that the difficulty can be traced to the fact that a variable can be eager normal form bisimilar to a non-variable, i.e., a  $\lambda$ -abstraction. In the proof below, we therefore first prove congruence of a “non-extensional” variant of  $\lambda\mu\rho$ -bisimilarity that does not relate variables to non-variables. This variant, called non- $\eta$  bisimilarity, is an extension of the first variant of pure eager normal form bisimilarity that we saw in the beginning of Section 2.3.

Then we recover the congruence result for full  $\lambda\mu\rho$ -bisimilarity, where variables *can* be related to non-variables, by means of a syntactic translation. This

translation is performed by applying an “infinite  $\eta$ -expansion” combinator reminiscent of Wadsworth’s  $J$  combinator from the example on page 339. The idea is that when showing that two translated terms are bisimilar, all values that one encounters will be  $\lambda$ -abstractions, not variables (they have been  $\eta$ -expanded). Therefore it does not matter whether one considers non- $\eta$  bisimilarity or full  $\lambda\mu\rho$ -bisimilarity.

In hindsight, the newer articles by Lassen and Levy [35] and Laird [27] shed some light on the complications in the congruence proof. Here is an intuitive explanation. Suppose that  $nt \sim nt'$  and  $v \sim v'$ , and that we want to prove that  $nt[v/x] \sim nt'[v'/x]$ . For simplicity we ignore stores. It turns out that we can evaluate  $nt[v/x]$  by alternating between evaluating the (open) terms  $nt$  and  $v$ , and by keeping track of pointers. For example, evaluation could proceed as follows:

- We begin by evaluating  $nt$  while remembering that  $x$  points to  $v$ .
- Suppose  $nt$  evaluates to  $NE_1[x v_1]$ . Since  $x$  points to  $v$ , we continue by evaluating  $[a]v y$  where  $a$  and  $y$  are fresh, remembering that  $a$  points to  $NE_1$  and that  $y$  points to  $v_1$ .
- Suppose  $[a]v y$  evaluates to  $[a]v_2$ . Since  $a$  points to  $NE_1$ , we continue by evaluating  $NE_1[z]$  where  $z$  is fresh, remembering that  $z$  points to  $v_2$ .

And so on. Notice that we alternate between evaluating terms that originate from  $nt$  and terms that originate from  $v$ : no non-trivial substitution is ever performed. Hence the assumptions that  $nt \sim nt'$  and  $v \sim v'$  can be applied fairly directly, and it is natural to attempt to prove that  $nt[v/x] \sim nt'[v'/x]$  by induction on the number of alternations in the evaluation above.

In the congruence proof in this article we instead prove that  $nt[v/x] \sim nt'[v'/x]$  by induction on the number of reduction steps starting from  $nt[v/x]$ . This number is not an entirely accurate replacement for the number of alternations as sketched above. Although some alternations naturally correspond to reductions in the substituted term—e.g., evaluating  $NE[x v]$  where  $x$  points to a  $\lambda$ -abstraction—not all of them do. A problem occurs when evaluating  $NE[x v]$  where  $x$  point to a variable  $y$ : there is no beta-reduction in the substituted term  $(NE[x v])[y/x]$  that corresponds to the alternation. Similarly when evaluating  $[a]v$  where  $a$  points to a simple named evaluation context  $[b][]$ . To rule out these two situations we essentially pre-process our terms by performing an infinite  $\eta$ -expansion: then variables never point to variables and names never point to simple named evaluation contexts.

The “alternation” intuition above follows the proof by Lassen and Levy [1] which is for a typed, but effect-free calculus. See also that article for a discussion of the relation between normal form bisimulation and pointer game semantics (in light of the “pointer-passing” semantics outlined above), and of earlier formalizations of such pointer-passing calculi related to game semantics. Laird [27] proves congruence of a simulation relation for a typed, stateful calculus. The proof relies on a modified semantics where arguments of function type are passed by pointer (see the second reduction rule on page 7 in that article). Jagadeesan et al. [21] prove congruence of a bisimulation relation for an aspect-oriented calculus which, as noted in the introduction, has state-like features. In that proof,  $\eta$ -expansion is

avoided by performing a separate induction on the length of “variable chains”: these can intuitively be understood in terms of the pointer-passing semantics outlined above, namely as chains of variables  $x_1, \dots, x_n$  such that  $x_i$  points to  $x_{i+1}$  for every  $i$ .

We do not know yet whether there exists a direct congruence proof, i.e., one that avoids  $\eta$ -expansion, which is based on a pointer-passing semantics and works for a calculus with the features of the  $\lambda\mu\rho$ -calculus. One benefit would be that one could then loosen up the syntax: in our congruence proof it is essential that computations are sequenced by means of let-bindings, since we rely on the  $\beta$ -reduction rule for let-bindings (the rule  $R\mu\rho 1$  on page 352) in order to count all alternations.

## 10.2 $\lambda\mu\rho$ -Bisimilarity Is an Equivalence Relation

We begin the congruence proof by showing that  $\lambda\mu\rho$ -bisimilarity is an equivalence relation. We also show some other preliminary results.

**Proposition 10.1.** *For every finite set  $J$  of references, there exists a greatest  $J$ -bisimulation  $\mathcal{B}_J$ .*

*Proof.* The definition of  $\mathcal{B}_J$  immediately implies that the union of an arbitrary family of  $J$ -bisimulations is also a  $J$ -bisimulation. In particular, the union of all  $J$ -bisimulations is the greatest  $J$ -bisimulation.  $\square$

At this point it is useful to change the definitions of a  $\lambda\mu\rho$ -bisimulation and of a  $J$ -bisimulation slightly: in those definitions, replace the operators  $V_{\mu\rho}$  and  $K_{\mu\rho}$  with  $V'_{\mu\rho}$  and  $K'_{\mu\rho}$ :

$$\begin{aligned} V'_{\mu\rho}(NR) &= \{(v, v') \mid \exists y \notin \text{FV}(v) \cup \text{FV}(v') . \\ &\quad \exists a \notin \text{FN}(v) \cup \text{FN}(v') . \\ &\quad (a \cdot y)[a]v y, [a]v' y) \in NR\}. \\ K'_{\mu\rho}(NR) &= \{(NE, NE') \mid \exists y \notin \text{FV}(NE) \cup \text{FV}(NE') . \\ &\quad (y|NE[y], NE'[y]) \in NR\}. \end{aligned}$$

These modifications do not change the relation of  $\lambda\mu\rho$ -bisimilarity; in fact, the greatest  $J$ -bisimulation is unchanged. The two operators  $V'_{\mu\rho}$  and  $K'_{\mu\rho}$  are more convenient in the congruence proof below, while the other two operators are more convenient when using  $\lambda\mu\rho$ -bisimulation as a proof principle.

We first show that  $\lambda\mu\rho$ -bisimilarity is an equivalence relation.

**Definition 10.2.** *Let  $NQ \subseteq NU_J$ .*

1.  *$NQ$  is closed under weakening if whenever  $(X_0|s, s', NR) \in NQ$  and  $X_0 \subseteq X$  for some finite set  $X$  of names and variables, also  $(X|s, s', NR) \in NQ$ .*
2.  *$NQ$  is closed under context extrusion if whenever  $(X|s, s', NR) \in NQ$  and  $(Z_1 \cdot Z_2|nt, nt') \in NR$ , then there exists  $NR' \supseteq NR \cup \{(Z_2|nt, nt')\}$  such that  $(X \cdot Z_1|s, s', NR') \in NQ$ .*

**Lemma 10.3.** *The greatest  $J$ -bisimulation is closed under weakening and context extrusion.*

**Lemma 10.4.**  $\lambda\mu\rho$ -bisimilarity is an equivalence relation.

*Proof (sketch).* Reflexivity and symmetry follow easily from the definition of  $B_J$ . As for transitivity, assume that  $t \sim_{\mu\rho} t'$  and that  $t' \sim_{\mu\rho} t''$ ; we must show that  $t \sim_{\mu\rho} t''$  (and similarly for named terms). Proposition 9.4 implies that there exists some  $J$  such that  $t$  and  $t'$  are  $J$ -bisimilar and  $t'$  and  $t''$  are  $J$ -bisimilar. Now consider a general composition construction on named term relation sets. Given  $NR_1 \subseteq NRel(Y, J, J_1)$  and  $NR_2 \subseteq NRel(Y, J, J_2)$ , define their composition as

$$NR_1; NR_2 = \{(X|nt_1, nt_2) \mid \exists nt. (X|nt_1, nt) \in NR_1 \text{ &} \\ (X|nt, nt_2) \in NR_2\},$$

and given  $NQ_1, NQ_2 \subseteq NU_J$ , define

$$NQ_1; NQ_2 = \{(X|s_1, s_2, NR_1; NR_2) \mid \\ \exists s. (X|s_1, s, NR_1) \in NQ_1 \& \\ (X|s, s_2, NR_2) \in NQ_2\}.$$

Then the following property holds: if  $NQ_1$  and  $NQ_2$  are  $J$ -bisimulations closed under weakening, then so is  $NQ_1; NQ_2$ .  $\square$

It remains to show that  $\lambda\mu\rho$ -bisimilarity is compatible. The proof of this fact is structured as follows, as outlined above:

- First, we show that a restricted variant of  $\lambda\mu\rho$ -bisimilarity is *substitutive* in a sense defined below. This variant, called non- $\eta$  bisimilarity, does not validate certain common extensionality rules for call-by-value calculi.
- Second, we use a syntactic translation to show that full  $\lambda\mu\rho$ -bisimilarity is substitutive. It follows that  $\lambda\mu\rho$ -bisimilarity is compatible.

### 10.3 Substitutions

A *substitution* is a finite map  $\sigma$  with a domain consisting of variables and names, and such that  $\sigma$  maps each variable in its domain to a  $\lambda\mu\rho$ -calculus value, and each name in its domain to a  $\lambda\mu\rho$ -calculus named evaluation context. Let  $\sigma$  range over substitutions. When  $\phi$  is a syntactic phrase (store, value, term, or named term),  $\phi\sigma$  denotes the result of “carrying out the substitution”  $\sigma$  on  $\phi$  (we omit the formal definitions). Also, define

$$NR(\sigma, \sigma') = \{(Z|nt\sigma, nt'\sigma') \mid (Z|nt, nt') \in NR\}$$

(where the variables and names occurring free in the ranges of  $\sigma$  and  $\sigma'$  are not in  $Z$ ).

Let  $dom(\sigma)$  denote the domain of  $\sigma$ . Say that

$$X \vdash \sigma \Sigma(NR) \sigma' : Y$$

when  $dom(\sigma) = dom(\sigma') = Y$ , and:

1. For every variable  $x \in Y$ ,  $(\sigma(x), \sigma'(x)) \in V'_{\mu\rho}(NR)$ .
2. For every name  $a \in Y$ ,  $(\sigma(a), \sigma'(a)) \in K'_{\mu\rho}(NR)$ .
3. The free variables and names in the ranges of  $\sigma$  and  $\sigma'$  are contained in  $X$ .

Say that two substitutions  $\sigma$  and  $\sigma'$  are  $\lambda\mu\rho$ -bisimilar (notation:  $\sigma \sim_{\mu\rho} \sigma'$ ) if there exists a  $J$ -bisimulation containing a tuple  $(X \cdot Y|\{\}, \{\}, NR)$  such that  $X \vdash \sigma \Sigma(NR) \sigma' : Y$ . In the next sections we show that  $\lambda\mu\rho$ -bisimilarity is substitutive in the following sense:

1. If  $t \sim_{\mu\rho} t'$  and  $\sigma \sim_{\mu\rho} \sigma'$ , then  $t\sigma \sim_{\mu\rho} t'\sigma'$ .
2. If  $nt \sim_{\mu\rho} nt'$  and  $\sigma \sim_{\mu\rho} \sigma'$ , then  $nt\sigma \sim_{\mu\rho} nt'\sigma'$ .

## 10.4 Non- $\eta$ Bisimulation

In order to show that  $\lambda\mu\rho$ -bisimilarity is substitutive, we first show the analogous result for a certain restricted variant of  $\lambda\mu\rho$ -bisimilarity. The variation consists in a change in the definition of the operators  $V$  and  $K$  (which are used to define relations on values and named evaluation contexts, respectively).

**Definition 10.5.** Let  $NR$  be a named term relation.

$$\begin{aligned} M^\dagger(NR) &= \{([a]v, [a]v'), (NE[x v], NE'[x v']) \mid \\ &\quad (v, v') \in V^\dagger(NR) \& (NE, NE') \in K^\dagger(NR)\} \\ V^\dagger(NR) &= \{(x, x) \mid x \text{ is a variable}\} \\ &\cup \{(\lambda x. t, \lambda x. t') \mid \exists a \notin \text{FN}(t) \cup \text{FN}(t'). \\ &\quad (x \cdot a | [a]t, [a]t') \in NR\} \\ K^\dagger(NR) &= \{([a][], [a][]) \mid a \text{ is a name}\} \\ &\cup \{(NE[\text{let } x = [] \text{ in } t], NE'[\text{let } x = [] \text{ in } t']) \mid \\ &\quad x \notin \text{FV}(NE) \cup \text{FV}(NE') \& \\ &\quad (x | NE[t], NE'[t']) \in NR\} \end{aligned}$$

## Definition 10.6

1. For every named term relation set  $NQ \subseteq NU_J$ , the named term relation set  $B_J^\dagger(NQ)$  is defined in the same way as  $B_J(NQ)$ , except that  $M^\dagger$  and  $V^\dagger$  are used instead of  $M_{\mu\rho}$  and  $V_{\mu\rho}$ .
2.  $NQ$  is a non- $\eta$   $J$ -bisimulation if  $NQ \subseteq B_J^\dagger(NQ)$ .
3. Two reference-closed  $\lambda\mu\rho$ -terms  $t$  and  $t'$  are non- $\eta$  bisimilar (written  $t \sim^\dagger t'$ ) if there exist a finite set of references  $J$  and a non- $\eta$   $J$ -bisimulation containing a tuple  $(X|\{\}, \{\}, NR)$  such that  $(t, t') \in T_{\mu\rho}(NR)$ . Non- $\eta$  bisimilarity of named terms is defined similarly.

The reason for the name “non- $\eta$ ” is that non- $\eta$  bisimilarity does not satisfy two common extensionality rules for call-by-value calculi, namely the  $\eta_v$ -rule and the  $\text{let}_\eta$ -rule:  $\lambda x. y \ x \not\sim^\dagger y$  and  $(\text{let } x = y \ z \text{ in } x) \not\sim^\dagger y \ z$ .

Let  $\mathcal{B}_J^\dagger$  be the greatest non- $\eta$   $J$ -bisimulation. The key to showing that non- $\eta$  bisimilarity is substitutive is to show that  $\mathcal{B}_J^\dagger$  is closed under substitutions in the sense defined next.

### Definition 10.7

1. For every  $NQ \subseteq NU_J$ , let

$$\begin{aligned} F^\dagger(NQ) = \{ & (X|s\sigma, s'\sigma', NR(\sigma, \sigma')) \mid \\ & \exists Y. (X \cdot Y|s, s', NR) \in NQ \text{ } \& \\ & X \vdash \sigma \Sigma^\dagger(NR) \sigma' : Y \} \end{aligned}$$

where  $\Sigma^\dagger(NR)$  is defined in the same way as  $\Sigma(NR)$ , except that  $V^\dagger$  and  $K^\dagger$  are used in place of  $V'_{\mu\rho}$  and  $K'_{\mu\rho}$  in the definition.

2. A named term relation set  $NQ \subseteq NU_J$  is closed under substitutions if  $F^\dagger(NQ) \subseteq NQ$ .

We now proceed to show that for every  $J$ , the greatest non- $\eta$   $J$ -bisimulation is closed under substitutions. Define the *substitutive closure* of  $NQ$  as

$$S^\dagger(NQ) = \bigcup_{n<\omega} (F^\dagger)^n(NQ).$$

It is the least fixed point of  $F^\dagger$  containing  $NQ$ .

**Main Lemma.** Let  $NQ \subseteq NU_J$  be a non- $\eta$   $J$ -bisimulation which is closed under context extrusion. Let  $(X|s, s', NR) \in (F^\dagger)^n(NQ)$  and  $(Z|nt, nt') \in NR$  and  $(v_j, v'_j) \in V^\dagger(NR)$  for all  $j \in J$ .

1. Assume that  $(\{j := v_j^{j \in J}\} \cdot s, nt) \mapsto^* (\{j := w_j^{j \in J}\} \cdot s_1, ne_1)$  in  $m$  or fewer steps. Then there exist  $X_1 \supseteq X \cdot Z$ ,  $s'_1$ ,  $ne'_1$ ,  $w_j'^{j \in J}$ , and  $NR_1 \supseteq NR$  such that

$$(\{j := v_j'^{j \in J}\} \cdot s', nt') \mapsto^* (\{j := w_j'^{j \in J}\} \cdot s'_1, ne'_1),$$

$(X_1|s_1, s'_1, NR_1) \in S^\dagger(NQ)$ ,  $(ne_1, ne'_1) \in M^\dagger(NR_1)$ , and also  $(w_j, w'_j) \in V^\dagger(NR_1)$  for all  $j \in J$ .

2. Conversely, assume that

$$(\{j := v_j'^{j \in J}\} \cdot s', nt') \mapsto^* (\{j := w_j'^{j \in J}\} \cdot s'_1, ne'_1)$$

in  $m$  or fewer steps. Then there exist  $X_1 \supseteq X \cdot Z$ ,  $s_1$ ,  $ne_1$ ,  $w_j^{j \in J}$ , and  $NR_1 \supseteq NR$  such that

$$(\{j := v_j^{j \in J}\} \cdot s, nt) \mapsto^* (\{j := w_j^{j \in J}\} \cdot s_1, ne_1)$$

etc.

*Proof (sketch).* By induction on the pairs  $(m, n)$ , ordered lexicographically. The proof is straightforward, but as noted above it does not generalize to full  $\lambda\mu\rho$ -bisimilarity. One needs the fact that variables cannot be related to non-variables, and a similar property for named evaluation contexts.  $\square$

**Corollary 10.8.** The greatest non- $\eta$   $J$ -bisimulation  $\mathcal{B}_J^\dagger$  is closed under substitutions.

*Proof (sketch).* The Main Lemma implies that

$$(F^\dagger)^n(\mathcal{B}_J^\dagger) \subseteq B_J^\dagger(S^\dagger(\mathcal{B}_J^\dagger))$$

for all  $n \geq 0$ . By definition of  $S^\dagger$  we then have  $S^\dagger(\mathcal{B}_J^\dagger) \subseteq B_J^\dagger(S^\dagger(\mathcal{B}_J^\dagger))$ . This means that  $S^\dagger(\mathcal{B}_J^\dagger)$  is a non- $\eta$   $J$ -bisimulation, and therefore  $F^\dagger(\mathcal{B}_J^\dagger) \subseteq S^\dagger(\mathcal{B}_J^\dagger) \subseteq \mathcal{B}_J^\dagger$ , since  $\mathcal{B}_J^\dagger$  is the largest non- $\eta$   $J$ -bisimulation.  $\square$

## 10.5 Non- $\eta$ Bisimilarity Is Substitutive

In order to show that non- $\eta$  bisimilarity is substitutive, one needs the following construction for combining named term relation sets:

**Definition 10.9.** Given  $NQ_1, NQ_2 \subseteq NU_J$ , define

$$\begin{aligned} NQ_1 + NQ_2 = \{ & (X|s_1 \cdot s_2, s'_1 \cdot s'_2, NR_1 \cup NR_2) \mid \\ & (X|s_1, s'_1, NR_1) \in NQ_1 \text{ \&} \\ & (X|s_2, s'_2, NR_2) \in NQ_2 \text{ \&} \\ & \text{dom}(s_1) \cap \text{dom}(s_2) = \\ & \text{dom}(s'_1) \cap \text{dom}(s'_2) = \emptyset \}. \end{aligned}$$

**Lemma 10.10.** If  $NQ_1$  and  $NQ_2$  are non- $\eta$   $J$ -bisimulations closed under weakening, then so is  $NQ_1 + NQ_2$ .

**Corollary 10.11.** The greatest non- $\eta$   $J$ -bisimulation  $\mathcal{B}_J^\dagger$  satisfies that  $\mathcal{B}_J^\dagger = \mathcal{B}_J^\dagger + \mathcal{B}_J^\dagger$ .

Finally, non- $\eta$  bisimilarity is substitutive:

## Theorem 10.12

1. If  $t \sim^\dagger t'$  and  $\sigma \sim^\dagger \sigma'$ , then  $t\sigma \sim^\dagger t'\sigma'$ .
2. If  $nt \sim^\dagger nt'$  and  $\sigma \sim^\dagger \sigma'$ , then  $nt\sigma \sim^\dagger nt'\sigma'$ .

*Proof (sketch)* We show the second implication—the first is completely similar. Assume that  $nt \sim^\dagger nt'$  and  $\sigma \sim^\dagger \sigma'$ , and let  $J$  be the set of free references in  $nt$ ,  $nt'$ ,  $\sigma$ , and  $\sigma'$ . Then the greatest non- $\eta$   $J$ -bisimulation  $\mathcal{B}_J^\dagger$  contains a tuple  $(X_1|\{\}, \{\}, NR_1)$  such that  $(|nt, nt') \in NR_1$  and a tuple  $(X_2 \cdot Y|\{\}, \{\}, NR_2)$  such that  $X_2 \vdash \sigma \Sigma^\dagger(NR_2) \sigma' : Y$ . Then by Corollary 10.11,  $\mathcal{B}_J^\dagger$  also contains the tuple  $(X_1 \cup X_2 \cup Y|\{\}, \{\}, NR_1 \cup NR_2)$ . Finally, since  $\mathcal{B}_J^\dagger$  is closed under substitutions, it also contains the tuple  $((X_1 \setminus Y) \cup X_2|\{\}, \{\}, NR_1(\sigma, \sigma') \cup NR_2(\sigma, \sigma'))$  where  $(|nt\sigma, nt\sigma') \in NR_1(\sigma, \sigma')$ . Hence  $nt\sigma \sim^\dagger nt'\sigma'$ .  $\square$

## 10.6 $\lambda\mu\rho$ -Bisimilarity Is Substitutive

The fact that  $\lambda\mu\rho$ -bisimilarity is substitutive can be derived from the analogous result for non- $\eta$  bisimilarity, Theorem 10.12, by means of a syntactic translation involving an “infinite  $\eta$ -expansion” combinator  $H$ .

Fix a finite set of references  $J = \{j_1, \dots, j_n\}$ . For every value  $v$  and every term  $t$ , define the term

$$\begin{aligned} \text{app}[v, t] = & \text{let } x_1 = !j_1 \text{ in } \dots \text{let } x_n = !j_n \text{ in} \\ & \text{let } y_1 = v \ x_1 \text{ in } \dots \text{let } y_n = v \ x_n \text{ in} \\ & (j_1 := y_1; \dots; j_n := y_n; \ t) \end{aligned}$$

(where  $x_1, \dots, x_n, y_1, \dots, y_n$  are not free in  $v$  or  $t$ ). The operational behavior of  $\text{app}[v, t]$  is to “apply  $v$  to every reference in  $J$ ” and then continue according to  $t$ . Now define

$$\begin{aligned} H_0 = & \lambda z. \lambda f. \lambda x. \text{let } y_1 = z \ x \text{ in} \\ & \text{app}[z, \text{let } y_2 = f \ y_1 \text{ in } \text{app}[z, z \ y_2]] \\ H = & \text{fix}[H_0]. \end{aligned}$$

The combinator  $H$  originates from a generalization of a “syntactic minimal invariance” equation [33, 48].

Also, for every value  $v$  and every named evaluation context  $NE$ , define

$$\begin{aligned} G[v] = & \lambda x. \text{let } y_1 = H \ x \text{ in} \\ & \text{app}[H, \text{let } y_2 = v \ y_1 \text{ in } \text{app}[H, H \ y_2]] \\ G[NE] = & NE[\text{let } x = [] \text{ in } \text{app}[H, H \ x]]. \end{aligned}$$

### Definition 10.13

1. For every term  $t$ , let  $t^\dagger$  be the result of substituting  $G[x]$  for every free variable  $x$  in  $t$ , and substituting  $G[[a][[]]]$  for every free name  $a$  in  $t$ . For every named term  $nt$ , define  $nt^\dagger$  similarly.
2. For every term  $t$ , define

$$t^\dagger = \text{app}[H, \text{let } x = t^\dagger \text{ in } \text{app}[H, H \ x]].$$

3. For every named term  $nt = [a]t$ , define  $nt^\dagger = [a]t^\dagger$ .

Using the above syntactic constructs,  $\lambda\mu\rho$ -bisimilarity can be characterized in terms of non- $\eta$  bisimilarity:

**Proposition 10.14.** *Let the free references of  $t$ ,  $t'$ ,  $v$ , and  $v'$  be contained in  $J$ .*

1.  $t \sim_{\mu\rho} t'$  iff  $t^\dagger \sim^\dagger t'^\dagger$ .
2.  $v \sim_{\mu\rho} v'$  iff  $G[v^\dagger] \sim^\dagger G[v'^\dagger]$ .
3.  $nt \sim_{\mu\rho} nt'$  iff  $nt^\dagger \sim^\dagger nt'^\dagger$ .

If  $v$  is a value such that the free references of  $v$  are contained in  $J$ , then we have  $v \sim_{\mu\rho} G[v]$ ; notice the analogy with the example about Wadsworth’s  $J$  combinator (page 339). In general, however,  $v \not\sim^\dagger G[v]$ . We show next that a more general property than  $v \sim_{\mu\rho} G[v]$  holds.

$$\begin{array}{c}
\frac{}{t \ R \ t} \quad \frac{v \ R \ v'}{v \ R \ G[v']} \quad \frac{t \ R \ t'}{\lambda x. t \ R \ \lambda x. t'} \\
\\
\frac{v_1 \ R \ v'_1 \quad v_2 \ R \ v'_2}{v_1 \ v_2 \ R \ v'_1 \ v'_2} \quad \frac{v \ R \ v' \quad t \ R \ t'}{(\iota:=v; t) \ R \ (\iota:=v'; t')} \\
\\
\frac{t \ R \ t' \quad \forall j \in J_0. s(j) \ R \ s'(j)}{\rho s. t \ R \ \rho s'. t'} \quad dom(s) = dom(s') = J_0, \\
J_0 \cap J = \emptyset \\
\\
\frac{nt \ R \ nt'}{\mu a. nt \ R \ \mu a. nt'} \quad \frac{}{NE \ R \ NE} \quad \frac{NE \ R \ NE'}{NE \ R \ G[NE']} \\
\\
\frac{t \ R \ t' \quad NE \ R \ NE'}{NE[\text{let } x=[] \text{ in } t] \ R \ NE'[\text{let } x=[] \text{ in } t']} \\
\\
\frac{NE \ R \ NE' \quad t \ R \ t'}{NE[t] \ R \ NE'[t']}
\end{array}$$


---

**Fig. 1.** The relation  $R$ 

**Definition 10.15.** *The binary relation  $R$  on terms, named evaluation contexts, and named terms is defined inductively by means of the inference rules in Figure 1. Two stores  $s$  and  $s'$  are related by  $R$  if they have the same domain  $J_0$  and if  $s(j) \ R \ s'(j)$  for all  $j \in J_0$ .*

**Proposition 10.16.** *Let  $J' \subseteq J$ . The named term relation set*

$$\{(X|s, s', NR) \in NU_{J'} \mid s \ R \ s' \ \& \ NR \subseteq \{(Z|nt, nt') \mid nt \ R \ nt'\}\}$$

*is a  $J'$ -bisimulation.*

In particular, taking  $J' = J$ :

**Corollary 10.17.** *Let the free references of  $t$ ,  $t'$ ,  $nt$ , and  $nt'$  be contained in  $J$ .*

1.  $t \ R \ t'$  implies  $t \sim_{\mu\rho} t'$ .
2.  $nt \ R \ nt'$  implies  $nt \sim_{\mu\rho} nt'$ .

It follows that  $\lambda\mu\rho$ -bisimilarity is substitutive:

**Theorem 10.18.**

1. If  $t \sim_{\mu\rho} t'$  and  $\sigma \sim_{\mu\rho} \sigma'$ , then  $t\sigma \sim_{\mu\rho} t'\sigma'$ .
2. If  $nt \sim_{\mu\rho} nt'$  and  $\sigma \sim_{\mu\rho} \sigma'$ , then  $nt\sigma \sim_{\mu\rho} nt'\sigma'$ .

*Proof (sketch).* As a simple example, assume that  $t \sim_{\mu\rho} t'$  and  $v \sim_{\mu\rho} v'$ ; it must be shown that  $t[v/x] \sim_{\mu\rho} t'[v'/x]$ . By Corollary 10.17,  $t[v/x] \sim_{\mu\rho} t[G[G[v]]/x]$ . Hence by Proposition 10.14 and the fact that non- $\eta$  bisimilarity is substitutive:

$$\begin{aligned}
(t[v/x])^\dagger &\sim^\dagger (t[G[G[v]]/x])^\dagger = t^\dagger[G[v^\ddagger]/x] \\
&\sim^\dagger t'^\dagger[G[v'^\ddagger]/x] \\
&\sim^\dagger (t'[v'/x])^\dagger.
\end{aligned}$$

By Proposition 10.14 again,  $t[v/x] \sim_{\mu\rho} t'[v'/x]$ .  $\square$

## 10.7 $\lambda\mu\rho$ -Bisimilarity Is a Congruence

Now we show that  $\lambda\mu\rho$ -bisimilarity is compatible, using the fact that it is substitutive.

**Proposition 10.19.**  $\mathcal{B}_J = \mathcal{B}_J + \mathcal{B}_J$ .

**Corollary 10.20.** Let  $\text{FR}(nt) \cup \text{FR}(nt') \subseteq J$ , and let  $y_j^{j \in J}$  be distinct variables not free in  $nt$  or  $nt'$ . Suppose that

$$\begin{aligned}
(\{j := y_j^{j \in J}\}, nt) &\mapsto^* (\{j := v_j^{j \in J}\} \cdot s_1, ne_1), \\
(\{j := y_j^{j \in J}\}, nt') &\mapsto^* (\{j := v'_j{}^{j \in J}\} \cdot s'_1, ne'_1),
\end{aligned}$$

and  $(X_1 | s_1, s'_1, NR_1) \in \mathcal{B}_J$  with  $(ne_1, ne'_1) \in M_{\mu\rho}(NR_1)$  and  $(v_j, v'_j) \in V_{\mu\rho}(NR_1)$  for all  $j \in J$ . Then  $nt \sim_{\mu\rho} nt'$ .

**Theorem 10.21.**  $\lambda\mu\rho$ -bisimilarity is compatible.

*Proof (sketch).* The most complicated case to show is that  $\lambda\mu\rho$ -bisimilarity is closed under  $\rho$ -abstraction: if  $t \sim_{\mu\rho} t'$  and also  $v_j \sim_{\mu\rho} v'_j$  for all  $j \in J_0$ , then

$$\rho\{j := v_j^{j \in J_0}\}. t \sim_{\mu\rho} \rho\{j := v'_j{}^{j \in J_0}\}. t'.$$

Here one proceeds in three steps:

1. If  $z \notin \text{FV}(t)$ , then  $\rho s. t \sim_{\mu\rho} \text{let } x = \rho s. \lambda z. t \text{ in } x!$ .
2. If  $v \sim_{\mu\rho} v'$  and  $v_j \sim_{\mu\rho} v'_j$  for all  $j \in J_0$ , and if the free references of all these values are contained in  $J \supseteq J_0$ , then

$$\rho\{j := G[v_j^\ddagger]^{j \in J_0}\}. G[v^\ddagger] \sim_{\mu\rho} \rho\{j := G[v'_j{}^\ddagger]^{j \in J_0}\}. G[v'^\ddagger].$$

3. If the free references of  $v$  and  $v_j^{j \in J_0}$  are contained in  $J \supseteq J_0$ , then

$$\rho\{j := v_j^{j \in J_0}\}. v \sim_{\mu\rho} \rho\{j := G[v_j^\ddagger]^{j \in J_0}\}. G[v^\ddagger].$$

The third part follows from Proposition 10.16 and Corollary 10.20. The proof of the second part uses Corollary 10.20 and the following construction: for every  $NQ \subseteq NU_J$  with  $J_0 \subseteq J$ , let  $NQ \setminus J_0$  be the subset of  $NU_{J \setminus J_0}$  defined by

$$\begin{aligned}
NQ \setminus J_0 = \{(X_1 | \{j := w_j^{j \in J_0}\} \cdot s, \{j := w'_j{}^{j \in J_0}\} \cdot s', NR) \mid \\
(X | s, s', NR) \in NQ \text{ \&} \\
X \subseteq X_1 \text{ \&} \\
\forall j \in J_0. (w_j, w'_j) \in V^\dagger(NR)\}.
\end{aligned}$$

The Main Lemma implies that  $\mathcal{B}_J^\dagger \setminus J_0 \subseteq \mathcal{B}_{J \setminus J_0}^\dagger$ .

The other cases of the proof are simpler and use Theorem 10.18 and Corollary 10.20.  $\square$

In summary, the main result of this section:

**Theorem 10.22.**  *$\lambda\mu\rho$ -bisimilarity is a congruence: an equivalence relation which is furthermore compatible.*

**Corollary 10.23.** *Each of  $\lambda$ -bisimilarity,  $\lambda\mu$ -bisimilarity, and  $\lambda\rho$ -bisimilarity is a congruence.*

*Proof.* It is easy to see that two  $\lambda\mu$ -terms are  $\lambda\mu$ -bisimilar if and only if they are  $\lambda\mu\rho$ -bisimilar, and similarly for the other inclusions between the four calculi considered in this article. (Each extension is “fully abstract”). The statement of the corollary immediately follows. Suppose for example that  $v_1 \sim_\mu v'_1$  and  $v_2 \sim_\mu v'_2$ . Then  $v_1 \sim_{\mu\rho} v'_1$  and  $v_2 \sim_{\mu\rho} v'_2$ . Therefore, since  $\lambda\mu\rho$ -bisimilarity is a congruence,  $v_1 v_2 \sim_{\mu\rho} v'_1 v'_2$ . Finally, since  $v_1 v_2$  and  $v'_1 v'_2$  are  $\lambda\mu$ -terms,  $v_1 v_2 \sim_\mu v'_1 v'_2$ .  $\square$

**Remark.** Non- $\eta$  bisimilarity is also a congruence. The relation between non- $\eta$  bisimilarity and  $\lambda\mu\rho$ -bisimilarity is analogous to the relation between Böhm tree equivalence and Böhm tree equivalence up to  $\eta$  for the pure  $\lambda$ -calculus.

## 11 Full Abstraction

In this section we show that  $\lambda\mu\rho$ -bisimilarity coincides with contextual equivalence for the  $\lambda\mu\rho$ -calculus.

First, let us say that a variable-closed and reference-closed named term *nt terminates*, written  $nt \Downarrow$ , iff  $\exists s, ne. (\{\}, nt) \mapsto^* (s, ne)$ . Then we define that terms  $t$  and  $t'$  are *contextually equivalent*, written  $t \cong_{\mu\rho} t'$ , iff for all names  $a$  and term contexts  $C$  such that  $C[t]$  and  $C[t']$  are variable-closed and reference-closed,  $[a]C[t] \Downarrow \Leftrightarrow [a]C[t'] \Downarrow$ . It is easy to see that  $\cong_{\mu\rho}$  is a congruence and, moreover, it is the largest congruence relation which satisfies that  $t \cong_{\mu\rho} t'$  implies  $[a]t \Downarrow \Leftrightarrow [a]t' \Downarrow$  for all names  $a$  and variable-closed and reference-closed terms  $t$  and  $t'$ . Since  $\lambda\mu\rho$ -bisimilarity is a congruence, it is immediate from its definition that it is included in contextual equivalence, viz. that  $\lambda\mu\rho$ -bisimilarity is sound with respect to contextual equivalence.

**Theorem 11.1 (Soundness).**  $\sim_{\mu\rho} \subseteq \cong_{\mu\rho}$ .

Similarly,  $\lambda$ -bisimilarity,  $\lambda\mu$ -bisimilarity, and  $\lambda\rho$ -bisimilarity are included in contextual equivalence for their respective calculi.

To prove the converse of Theorem 11.1, we will form a bisimulation which relates all contextually equivalent terms. We first motivate our approach by an informal discussion of a few examples.

First, recall the  $\lambda$ -terms  $x I (x I I)$  and  $x I (x \Omega I)$  from Section 2.2. We noted then that these two terms are not head normal form bisimilar, and that a distinguishing context should intuitively provide a function  $x$  that first selects its second argument and then selects its first argument. In the  $\lambda\mu\rho$ -calculus counterpart of this example, such an  $x$  can actually be defined: a function can use local state to do different things each time it is called. Borrowing terminology from game semantics, we associate with each free variable and each free name a certain *strategy* for interaction with its environment.

Consider now the pair of terms from the example in Section 5:

$$\begin{aligned} t &= \rho\{\iota:=I\}. \text{let } z=x (\lambda y. (!\iota) y) \text{ in } (\iota:=\lambda z.\Omega; x_0 I) \\ t' &= \text{let } z=x (\lambda y. I y) \text{ in } x_0 I \end{aligned}$$

We have seen that these two terms are not eager normal form bisimilar. Intuitively, after executing the assignment  $\iota := \lambda z.\Omega$ , the two values  $\lambda y. (!\iota) y$  and  $\lambda y. I y$  can no longer match.

What context  $C$  would distinguish  $t$  and  $t'$ ? One idea is as follows: the context should supply a function  $x$  and a function  $x_0$  that have access to a common reference  $\jmath$ . The strategy for  $x$  should be to save its argument in  $\jmath$ . The strategy for  $x_0$  should be to read the function in  $\jmath$  and apply it to, say,  $I$ . This gives the following definition:

$$C = \text{let } x=(\lambda y_0. \jmath:=y_0; I) \text{ in let } x_0=(\lambda y_1. \text{let } z_1=\jmath \text{ in } z_1 I) \text{ in []}$$

Indeed, the configuration  $(\{\}, C[t])$  diverges while the configuration  $(\{\}, C[t'])$  converges.

More generally, the calculations in the example in Section 5 suggest that in order to show that two terms are not eager normal form bisimilar, one must calculate a number of pairs of reduction sequences. One starts reducing the two original terms (together with the empty store). After that, one can at each step choose to consider any pair of evaluation contexts or any pair of values that one has at some point seen was required to match. This method can in fact be mimicked by terms implementing pre-programmed strategies. All strategies share a single, “global” reference containing a list. Every function that implements a strategy adds every single argument it sees to that list, generalizing the way that the “ $x$ ” above stored its argument  $\lambda y. (!\iota) y$ . Furthermore, every such function also adds every *evaluation context* it sees to the global list; this is done using the expressive power of the  $\lambda\mu$ -calculus. Now, every other strategy, such as the “ $x_0$ ” above, can at any point retrieve any previously seen value or evaluation context from the global list.

We now turn to the formal proof. We will need to accumulate values in lists and random access list entries. We encode the empty list as the identity function  $\mathbf{I}$  and we encode the list  $v$  with elements  $v_1, \dots, v_n$  appended as:

$$\begin{aligned} \langle v | v_1, \dots, v_n \rangle &= \\ &\lambda z. \text{let } x_0=v z \text{ in let } x_1=x_0 v_1 \text{ in } \dots \text{ in } x_{n-1} v_n \end{aligned}$$

where  $z, x_0, \dots, x_{n-1}$  are not free in  $v, v_1, \dots, v_n$ . When  $v = \mathbb{I}$ , we write just  $\langle v_1, \dots, v_n \rangle$ . We access the  $q$ 'th element in the list  $v$  with

$$v\#q = \mu a. [a]v \lambda x_1 \dots \lambda x_{q-1}. \hat{a}$$

where  $a$  is not free in  $v$ .

We use a designated reference  $j_0$  to store a list with all the arguments  $v$  and continuations  $NE$  we see along the way (each  $NE$  is stored as the value  $\lambda x. \mu a. NE[x]$  with  $x, a$  not free in  $NE$ ). We let  $w$  range over both pairs of values and pairs of named evaluation contexts. Given such a pair, let  $w^V$  be the pair of values and let  $w^{NR}$  be the singleton named term relation defined as:

$$w^V = \begin{cases} (v_1, v_2) & \text{if } w = (v_1, v_2) \\ (\lambda x. \mu a. NE_1[x], \lambda x. \mu a. NE_2[x]) & \text{if } w = (NE_1, NE_2) \end{cases}$$

$$w^{NR} = \begin{cases} (a \cdot x | [a](v_1 \star x), [a](v_2 \star x)) & \text{if } w = (v_1, v_2) \\ (x | NE_1 \star x, NE_2 \star x) & \text{if } w = (NE_1, NE_2) \end{cases}$$

where  $x$  and  $a$  are not free in  $v_1, v_2, NE_1, NE_2$ .

For each free variable  $x_i$  and name  $a_j$ , we associate a reference  $i$  and  $j$ , respectively. Each reference stores a *strategy*  $p$  which is a sequence of *moves*  $m$ , one for each successive invocation of the variable  $x_i$  or name  $a_j$ , ending in “success”  $\top$  or “failure”  $\perp$ :

$$\begin{aligned} \text{STRATEGIES } p ::= & \top \mid \perp \mid m; p \\ \text{MOVES } m ::= & \text{move}(q, p_1, p_2) \quad (q \geq 1) \end{aligned}$$

A move  $\text{move}(q, p_1, p_2)$  “plays” the  $q$ 'th value or continuation from the list stored in  $j_0$  and associates the strategy  $p_1$  with a variable, which is used as argument, and  $p_2$  with a name, which is used as continuation. These ideas are expressed in the following encodings. Given a reference  $i$  and a strategy  $p$ , we define the function  $\llbracket p \rrbracket(i)$ , which takes a function argument  $x$  and a continuation  $y$  as arguments, and for every move  $m$ , we define the term  $\llbracket m \rrbracket$ , as follows.

$$\begin{aligned} \llbracket \top \rrbracket(i) &= \lambda x. \lambda y. \mathbb{I} \\ \llbracket \perp \rrbracket(i) &= \lambda x. \lambda y. \Omega \\ \llbracket m; p \rrbracket(i) &= \lambda x. \lambda y. \text{let } z = !j_0 \text{ in } j_0 := \langle z | x, y \rangle; \upsilon := \llbracket p \rrbracket(i); \llbracket m \rrbracket \\ \llbracket \text{move}(q, p_1, p_2) \rrbracket &= \\ &\text{let } z = !j_0 \text{ in let } x = z \# q \text{ in } \rho \{ \iota := \llbracket p_1 \rrbracket(i), j := \llbracket p_2 \rrbracket(j) \}. \check{j}[x \hat{i}] \end{aligned}$$

where  $\hat{i}$  and  $\check{j}$  denote the value and the evaluation context

$$\begin{aligned} \hat{i} &= \lambda x. \mu a. [a_0] \text{let } y = !\iota \text{ in let } z = y x \text{ in } z \hat{a} \\ \check{j} &= \text{let } x = [] \text{ in let } y = !j \text{ in let } z = y x \text{ in } z \hat{a}_0 \end{aligned}$$

and where  $a_0$  is a designated “top-level” name.

These are the building blocks we use to “separate” terms. As in [32], our separation proof is co-inductive. We define the named term relation set  $\mathcal{W}$  in Figure 2.

$$\begin{aligned}
\mathcal{W} = & \{(x_1, \dots, x_m, a_1, \dots, a_n | s_1, s_2, \{w_1^{\text{NR}}, \dots, w_q^{\text{NR}}\}) \\
& \mid \forall \text{ strategies } p_1, \dots, p_m, p'_1, \dots, p'_n. \\
& \quad \forall \text{ distinct references } \iota_1, \dots, \iota_m, j_1, \dots, j_n \\
& \quad \notin \text{dom}(s_1) \cup \text{dom}(s_2). \\
& \quad \forall \text{ moves } m. \\
& \quad [a_0]\rho s \cdot s'_1\sigma. \llbracket m \rrbracket \Downarrow \Leftrightarrow [a_0]\rho s \cdot s'_2\sigma. \llbracket m \rrbracket \Downarrow \\
& \quad \text{where } s = \{\iota_1 := \llbracket p_1 \rrbracket(\iota_1), \dots, \iota_m := \llbracket p_m \rrbracket(\iota_m), \\
& \quad \quad j_1 := \llbracket p'_1 \rrbracket(j_1), \dots, j_n := \llbracket p'_n \rrbracket(j_n)\}, \\
& \quad \sigma = [\hat{i}_1/x_1, \dots, \hat{i}_m/x_m, [a_0]j_1/a_1, \dots, [a_0]j_n/a_n], \\
& \quad s'_i = s_i \cdot \{j_0 := \langle v_{i1}, \dots, v_{iq} \rangle\}, \\
& \quad (v_{11}, v_{21}) = w_1^{\vee}, \dots, (v_{1q}, v_{2q}) = w_q^{\vee}\}
\end{aligned}$$

**Fig. 2.** The named term relation set  $\mathcal{W}$

**Lemma 11.2.**  $\mathcal{W}$  is a  $\lambda\mu\rho$ -bisimulation.

*Proof.* By detailed analysis of the possible operational behaviours of the named terms in each  $w_i^{\text{NR}}$  triple.  $\square$

**Lemma 11.3.**  $t_1 \cong_{\mu\rho} t_2$  implies  $t_1 \sim_{\mu\rho} t_2$  if  $\text{FR}(t_1) \cup \text{FR}(t_2) = \emptyset$ .

*Proof.* Suppose  $t_1$  and  $t_2$  are reference-closed terms,  $t_1 \cong_{\mu\rho} t_2$ , and  $X \vdash t_1, t_2$ . Then, by the definition of  $\mathcal{W}$  and contextual equivalence,  $t_1 \cong_{\mu\rho} t_2$  implies  $(X|\{\}, \{\}, \{w^{\text{NR}}\}) \in \mathcal{W}$  where  $w = (\lambda x. t_1, \lambda x. t_2)$  and  $x \notin X$ . Observe that  $w^{\text{NR}} = (a[x][a]t_1, [a]t_2)$ . We conclude  $t_1 \sim_{\mu\rho} t_2$  by Lemmas 11.2 and 10.3.  $\square$

To extend this result to general terms, we define a term context  $L_J$  that “translates” any term  $t$  with  $\text{FV}(t) \subseteq J = \{\iota_1, \dots, \iota_n\}$  to the reference-closed closed term

$$\begin{aligned}
L_J[t] = & \lambda x. \rho \{\iota_1 := !, \dots, \iota_n := !\}. \\
& \langle \lambda x. t, \text{get}(\iota_1), \text{set}(\iota_1), \dots, \text{get}(\iota_n), \text{set}(\iota_n) \rangle
\end{aligned}$$

where  $x \notin \text{FV}(t)$ ,  $\text{get}(\iota_i) = \lambda x. !\iota_i$ , and  $\text{set}(\iota_i) = \lambda x. (\iota_i := x; !)$ .

**Lemma 11.4.**  $t_1 \sim_{\mu\rho} t_2$  iff  $L_J[t_1] \sim_{\mu\rho} L_J[t_2]$ .

**Theorem 11.5 (Completeness).**  $\sim_{\mu\rho} \supseteq \cong_{\mu\rho}$ .

*Proof.* Suppose  $J = \text{FR}(t_1) \cup \text{FR}(t_2)$ . Then

$$\begin{aligned}
t_1 \cong_{\mu\rho} t_2 & \Rightarrow L_J[t_1] \cong_{\mu\rho} L_J[t_2] & \cong_{\mu\rho} \text{ is a congruence} \\
& \Rightarrow L_J[t_1] \sim_{\mu\rho} L_J[t_2] & \text{Lemma 11.3} \\
& \Rightarrow t_1 \sim_{\mu\rho} t_2 & \text{Lemma 11.4.}
\end{aligned}$$

$\square$

## Acknowledgements

We thank Olivier Danvy and Andrzej Filinski for guidance and comments on this work. We also thank the anonymous referees as well as Matthias Felleisen and the POPL reviewers for comments and suggestions, and Lars Birkedal, Nina

Bohr, Radha Jagadeesan, Vasileios Koutavas, Paul Levy, Corin Pitcher, and Eijiro Sumii for discussions.

## References

- [1] Abramsky, S.: The lazy lambda calculus. In: Turner, D. (ed.) Research Topics in Functional Programming, pp. 65–116. Addison-Wesley, Reading (1990)
- [2] Abramsky, S., Honda, K., McCusker, G.: A fully abstract game semantics for general references. In: Pratt, V. (ed.) Proceedings of the Thirteenth Annual IEEE Symposium on Logic in Computer Science, June 1998, pp. 334–344. IEEE Computer Society Press, Los Alamitos (1998)
- [3] Ahmed, A., Dreyer, D., Rossberg, A.: State-dependent representation independence. In: Shao, Z., Pierce, B.C. (eds.) Proceedings of the Thirty-Sixth Annual ACM Symposium on Principles of Programming Languages, pp. 340–353. ACM Press, New York (2009)
- [4] Ahmed, A.J.: Step-indexed syntactic logical relations for recursive and quantified types. In: Sestoft [56], pp. 69–83
- [5] Appel, A.W., McAllester, D.A.: An indexed model of recursive types for foundational proof-carrying code. ACM Transactions on Programming Languages and Systems 23(5), 657–683 (2001)
- [6] Ariola, Z.M., Herbelin, H.: Minimal classical logic and control operators. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 871–885. Springer, Heidelberg (2003)
- [7] Barendregt, H.: The Lambda Calculus: Its Syntax and Semantics. Studies in Logic and the Foundation of Mathematics, vol. 103. North-Holland, Amsterdam (1984) (revised edn.)
- [8] Bierman, G.M.: A computational interpretation of the  $\lambda\mu$ -calculus. In: Brim, L., Gruska, J., Zlatuška, J. (eds.) MFCS 1998. LNCS, vol. 1450, pp. 336–345. Springer, Heidelberg (1998)
- [9] Boudol, G.: On the semantics of the call-by-name CPS transform. Theoretical Computer Science 234(1–2), 309–321 (2000)
- [10] Danvy, O., Nielsen, L.R.: Refocusing in reduction semantics. Research Report BRICS RS-04-26, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark (November 2004); A preliminary version appears in the informal proceedings of the Second International Workshop on Rule-Based Programming (RULE 2001), Electronic Notes in Theoretical Computer Science, vol. 59.4
- [11] David, R., Py, W.:  $\lambda\mu$ -calculus and Böhm’s theorem. Journal of Symbolic Logic 66(1), 407–413 (2001)
- [12] de Groote, P.: On the relation between the lambda-mu-calculus and the syntactic theory of sequential control. In: Pfenning, F. (ed.) LPAR 1994. LNCS, vol. 822, pp. 31–43. Springer, Heidelberg (1994)
- [13] Egidi, L., Honsell, F., Ronchi della Rocca, S.: Operational, denotational and logical descriptions: a case study. Fundamenta Informaticae 16(2), 149–169 (1992)
- [14] Felleisen, M.:  $\lambda$ -v-CS: An extended  $\lambda$ -calculus for scheme. In: Proceedings of the 1988 ACM Conference on LISP and Functional Programming, pp. 72–85 (1988)
- [15] Felleisen, M., Hieb, R.: The revised report on the syntactic theories of sequential control and state. Theoretical Computer Science 103, 235–271 (1992)

- [16] Friedman, D.P., Haynes, C.T.: Constraining control. In: Deusen, M.S.V., Galil, Z. (eds.) *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, New Orleans, Louisiana, January 1985, pp. 245–254. ACM Press, New York (1985)
- [17] Gordon, A.D.: Bisimilarity as a theory of functional programming. *Theoretical Computer Science* 228(1–2), 5–47 (1999)
- [18] Gordon, A.D., Pitts, A.M. (eds.): *Higher Order Operational Techniques in Semantics*. Publications of the Newton Institute, Cambridge University Press (1998)
- [19] Howe, D.J.: Proving congruence of bisimulation in functional programming languages. *Information and Computation* 124(2), 103–112 (1996)
- [20] Jacobs, B., Rutten, J.: A tutorial on (co)algebras and (co)induction. *Bulletin of the European Association for Theoretical Computer Science* 62, 222–259 (1997)
- [21] Jagadeesan, R., Pitcher, C., Riely, J.: Open bisimulation for aspects (2009) (Long version, to appear)
- [22] Kelsey, R., Clinger, W., Rees, J. (eds.): Revised<sup>5</sup> report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation* 11(1), 7–105 (1998)
- [23] Koutavas, V., Wand, M.: Bisimulations for untyped imperative objects. In: Sestoft [56], pp. 146–161
- [24] Koutavas, V., Wand, M.: Reasoning about class behavior. In: FOOL/WOOD 2007 Workshop (January 2007)
- [25] Koutavas, V., Wand, M.: Small bisimulations for reasoning about higher-order imperative programs. In: Peyton Jones, S. (ed.) *Proceedings of the Thirty-Third Annual ACM Symposium on Principles of Programming Languages*, Charleston, South Carolina, January 2006. SIGPLAN Notices, vol. 41(1), pp. 141–152. ACM Press, New York (2006)
- [26] Laird, J.: Full abstraction for functional languages with control. In: Winskel, G. (ed.) *Proceedings of the Twelfth Annual IEEE Symposium on Logic in Computer Science*, Warsaw, Poland, June 1997, pp. 58–67. IEEE Computer Society Press, Los Alamitos (1997)
- [27] Laird, J.: A fully abstract trace semantics for general references. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) *ICALP 2007. LNCS*, vol. 4596, pp. 667–679. Springer, Heidelberg (2007)
- [28] Landin, P.J.: The mechanical evaluation of expressions. *The Computer Journal* 6(4), 308–320 (1964)
- [29] Lassen, S.B.: Bisimulation up to context for imperative lambda calculus. Unpublished note. Presented at The Semantic Challenge of Object-Oriented Programming, Schloss Dagstuhl (1998)
- [30] Lassen, S.B.: Eager normal form bisimulation. In: Panangaden, P. (ed.) *Proceedings of the Twentieth Annual IEEE Symposium on Logic in Computer Science*, June 2005, pp. 345–354. IEEE Computer Society Press, Los Alamitos (2005)
- [31] Lassen, S.B.: Normal form simulation for McCarty’s amb. In: Escardó, M., Jung, A., Mislove, M. (eds.) *Proceedings of the 21st Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXI)*, May 2005. Electronic Notes in Theoretical Computer Science, vol. 155, pp. 445–465. Elsevier Science Publishers, Amsterdam (2005)
- [32] Lassen, S.B.: Head normal form bisimulation for pairs and the  $\lambda\mu$ -calculus (extended abstract). In: Alur, R. (ed.) *Proceedings of the Twenty-First Annual IEEE Symposium on Logic in Computer Science*, August 2006, pp. 297–306. IEEE Computer Society Press, Los Alamitos (2006)
- [33] Lassen, S.B.: Relational reasoning about contexts. In: Gordon and Pitts [18], pp. 91–135

- [34] Lassen, S.B.: Bisimulation in untyped lambda calculus: Böhm trees and bisimulation up to context. In: Brookes, S., Jung, A., Mislove, M., Scedrov, A. (eds.) Proceedings of the 15th Annual Conference on Mathematical Foundations of Programming Semantics, April 1999. Electronic Notes in Theoretical Computer Science, vol. 20, pp. 346–374. Elsevier Science Publishers, Amsterdam (1999)
- [35] Lassen, S.B., Levy, P.B.: Typed normal form bisimulation. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS, vol. 4646, pp. 283–297. Springer, Heidelberg (2007)
- [36] Lassen, S.B., Levy, P.B.: Typed normal form bisimulation for parametric polymorphism. In: Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, pp. 341–352. IEEE Computer Society Press, Los Alamitos (2008)
- [37] Levy, P.B.: Game semantics using function inventories. Talk given at Geometry of Computation 2006, Marseille (2006)
- [38] Mason, I.A., Talcott, C.L.: Equivalence in functional languages with effects. Journal of Functional Programming 1(3), 297–327 (1991)
- [39] Merro, M., Biasi, C.: On the observational theory of the CPS-calculus (extended abstract). In: Brookes, S., Mislove, M. (eds.) Proceedings of the 22nd Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXII), May 2006. Electronic Notes in Theoretical Computer Science, vol. 158, pp. 307–330. Elsevier Science Publishers, Amsterdam (2006)
- [40] Meyer, A.R., Sieber, K.: Towards fully abstract semantics for local variables: Preliminary report. In: Ferrante, J., Mager, P. (eds.) Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, January 1988, pp. 157–169. ACM Press, New York (1988)
- [41] Milner, R., Tofte, M., Harper, R., MacQueen, D.: The Definition of Standard ML (Revised). The MIT Press, Cambridge (1997)
- [42] Mosses, P.D.: Component-based description of programming languages. In: Visions of Computer Science, Proc. BCS International Academic Research Conference, pp. 275–286. BCS, London (2008)
- [43] Mosses, P.D.: Action Semantics. Cambridge Tracts in Theoretical Computer Science, vol. (26). Cambridge University Press, Cambridge (1992)
- [44] Mosses, P.D.: Theory and practice of action semantics. In: Penczek, W., Szałas, A. (eds.) MFCS 1996. LNCS, vol. 1113, pp. 37–61. Springer, Heidelberg (1996)
- [45] Parigot, M.:  $\lambda\mu$ -calculus: an algorithmic interpretation of classical natural deduction. In: Voronkov, A. (ed.) LPAR 1992. LNCS, vol. 624, pp. 190–201. Springer, Heidelberg (1992)
- [46] Perez, R.P.: An extensional partial combinatory algebra based on  $\lambda$ -terms. In: Tarlecki, A. (ed.) MFCS 1991. LNCS, vol. 520, pp. 387–396. Springer, Heidelberg (1991)
- [47] Pitts, A.M.: Reasoning about local variables with operationally-based logical relations. In: O’Hearn, P.W., Tennent, R.D. (eds.) Algol-Like Languages, ch. 17, vol. 2, pp. 173–193. Birkhauser, Basel (1997); Reprinted from Proceedings Eleventh Annual IEEE Symposium on Logic in Computer Science, pp. 152–163 (1996)
- [48] Pitts, A.M., Stark, I.D.B.: Operational reasoning for functions with local state. In: Gordon and Pitts [18], pp. 227–273
- [49] Plotkin, G.D.: Call-by-name, call-by-value and the  $\lambda$ -calculus. Theoretical Computer Science 1, 125–159 (1975)
- [50] Ritter, E., Pitts, A.M.: A fully abstract translation between a  $\lambda$ -calculus with reference types and Standard ML. In: Dezani-Ciancaglini, M., Plotkin, G. (eds.) TLCA 1995. LNCS, vol. 902, pp. 397–413. Springer, Heidelberg (1995)

- [51] Sangiorgi, D.: Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms. PhD thesis, University of Edinburgh (1992); LFCS report ECS-LFCS-93-266 (also published as CST-99-93)
- [52] Sangiorgi, D.: A theory of bisimulation for the pi-calculus. *Acta Informatica* 33, 69–97 (1996),  
<ftp://ftp-sop.inria.fr/mimosa/personnel/davides/sub.ps.gz>
- [53] Sangiorgi, D.: Lazy functions and mobile processes. In: Proof, language, and interaction: essays in honour of Robin Milner, pp. 691–720. MIT Press, Cambridge (2000)
- [54] Sangiorgi, D.: The lazy lambda calculus in a concurrency scenario. *Information and Computation* 111(1), 120–153 (1994)
- [55] Sangiorgi, D., Kobayashi, N., Sumii, E.: Environmental bisimulations for higher-order languages. In: Ong, C.L. (ed.) Proceedings of the Twenty-Second Annual IEEE Symposium on Logic in Computer Science, Wroclaw, Poland, July 2007, pp. 293–302. IEEE Computer Society Press, Los Alamitos (2007)
- [56] Sestoft, P. (ed.): ESOP 2006. LNCS, vol. 3924. Springer, Heidelberg (2006)
- [57] Størvring, K.: On Reasoning Equationally: Lambda Calculi and Programs with Computational Effects. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark (August 2007)
- [58] Størvring, K., Lassen, S.B.: A complete, co-inductive syntactic theory of sequential control and state. In: Felleisen, M. (ed.) Proceedings of the Thirty-Fourth Annual ACM Symposium on Principles of Programming Languages, Nice, France, January 2007. SIGPLAN Notices, vol. 42(1), pp. 161–172. ACM Press, New York (2007)
- [59] Sumii, E.: A theory of non-monotone memory (or: Contexts for free). In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 237–251. Springer, Heidelberg (2009)
- [60] Sumii, E.: A complete characterization of observational equivalence in polymorphic lambda-calculus with general references (manuscript) (January 2009)
- [61] Sumii, E., Pierce, B.C.: A bisimulation for type abstraction and recursion. In: Abadí, M. (ed.) Proceedings of the Thirty-Second Annual ACM Symposium on Principles of Programming Languages, Long Beach, California, January 2005. SIGPLAN Notices, vol. 40(1), pp. 63–74. ACM Press, New York (2005)
- [62] Talcott, C.: Reasoning about functions with effects. In: Gordon and Pitts [18], pp. 347–390
- [63] Vytiniotis, D., Koutavas, V.: Relating step-indexed logical relations and bisimulations. Technical Report MSR-TR-2009-25, Microsoft Research, Cambridge (March 2009)
- [64] Wand, M., Sullivan, G.T.: Denotational semantics using an operationally-based term model. In: Jones, N.D. (ed.) Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Programming Languages, France, January 1997, pp. 386–399. ACM Press, New York (1997)

# Vertical Object Layout and Compression for Fixed Heaps

Ben L. Titzer<sup>1</sup> and Jens Palsberg<sup>2</sup>

<sup>1</sup> Sun Microsystems Laboratories, Menlo Park

<sup>2</sup> UCLA, University of California, Los Angeles

**Abstract.** Research into embedded sensor networks has placed increased focus on the problem of developing reliable and flexible software for microcontroller-class devices. Languages such as nesC [10] and Virgil [20] have brought higher-level programming idioms to this lowest layer of software, thereby adding expressiveness. Both languages are marked by the absence of dynamic memory allocation, which removes the need for a runtime system to manage memory. While nesC offers code modules with statically allocated fields, arrays and structs, Virgil allows the application to allocate and initialize arbitrary objects during compilation, producing a fixed object heap for runtime. This paper explores techniques for compressing fixed object heaps with the goal of reducing the RAM footprint of a program. We explore table-based compression and introduce a novel form of object layout called *vertical object layout*. We provide experimental results that measure the impact on RAM size, code size, and execution time for a set of Virgil programs. Our results show that compressed vertical layout has better execution time and code size than table-based compression while achieving more than 20% heap reduction on 6 of 12 benchmark programs and 2–17% heap reduction on the remaining 6. We also present a formalization of vertical object layout and prove tight relationships between three styles of object layout.

## 1 Introduction

Microcontrollers are tiny, low-power embedded processors deployed to control and monitor consumer products from microwaves, to fuel-injection systems, to sensor networks. A typical microcontroller integrates a central processing unit, RAM, reprogrammable flash memory, and IO devices on a single chip. They have very limited computational power, and their main memory is often measured in hundreds of bytes to a few kilobytes. For example, a popular microcontroller unit from Atmel, the ATMega128, has 4KB of RAM and 128KB of flash to store code and read-only data; it serves as the central processing unit of the Mica2 sensor network node.

The primary resource limitation in many microcontroller applications is the RAM space available for storing the program heap and stack. This often precludes a dynamic memory management system, either a manual one such as C’s `malloc()` and `free()`, or an automatic system such as a garbage collector. For

that reason, many microcontroller applications are written to pre-allocate all of their needed data structures and do not use any form of dynamic memory allocation. In the C programming language, this is done by statically declaring arrays and structures as global variables. In nesC [10], modules may contain state represented as fields, and are instantiated by a module-wiring system at compile time. In Virgil [20], applications allocate and initialize their heap at compilation time by running code in an interpreter built into the compiler that supports the complete language.

Reducing RAM consumption allows larger applications to be built and deployed on the same microcontroller model, while reducing the resource requirements of a single application allows a smaller, cheaper microcontroller to accomplish the same task. There are several general techniques to reduce RAM consumption. One is to simply remove unused data structures through compiler or manual analysis. Another is to reduce the average footprint of a program by employing virtual memory techniques that move infrequently used data to larger, slower storage such as disk. A third is to compress infrequently used data and dynamically decompress it as it is accessed. A fourth technique, employed here, is to statically compress program quantities so that dynamic decompression is unnecessary. We discuss previous techniques in more detail in the related work section.

This paper evaluates two offline heap-compression techniques for programs written in the Virgil language. Both techniques exploit the availability of the entire program heap at compile time and are employed on top of our compiler that already performs sophisticated dead code and data elimination. The first compression technique, table-based reference compression, was described in our previous work [20] and is evaluated in more detail here. The second technique is a novel object layout model that we call *vertical object layout*. Vertical object layout represents objects in a more compact way by viewing the heap as a collection of field arrays that are indexed by object number, rather than the traditional approach of a collection of objects that are accessed via pointers. Object numbers can then be compressed without additional indirections. We present a simple object-numbering system for identifiers that ensures that each field array is stored without wasting space, even in the presence of subclassing. Our experimental results show that vertical object layout has better execution time and code size than the table-based compression scheme on nearly all benchmarks, while achieving similar RAM size savings. Relative to the standard object layout strategy, the code size increase from vertical layout is less than 10% for most programs, and less than 15% for all programs, while the execution time overhead is less than 10% for 7 of 12 programs, less than 20% for 9, and between 30% and 90% for the remaining 3. Interestingly, compressed vertical layout provides a more efficient typecast operation, which actually improves execution time over the baseline for two programs that use dynamic casts intensively.

We first presented vertical object layout in a preliminary version [23] of this paper at CASES 2007; the present paper has improved explanations and a formalization of vertical object layout. Our formalization leads to a theorem that a

semantics based on the usual *horizontal* object layout is equivalent to a semantics based on vertical object layout.

The outline of this paper is as follows. Section 2 gives background for this paper by describing the key features of Virgil that are relevant to this work. Section 3 describes the basic idea of reference compression, summarizing our previous work in the area. Section 4 describes our new object model and its application to reference compression. Section 5 presents a formalization of vertical object layout. Section 6 gives our experimental results. Section 7 describes related work, and Section 8 gives our conclusion and vision of future work.

## 2 Virgil Background

In our previous work [20], we have developed the Virgil programming language and compiler system that targets microcontroller-class devices. Virgil is a light-weight object-oriented language that is closest to Java, C++ and C#. Like Java, Virgil provides single inheritance between classes, with all methods virtual by default, except those declared private. Objects are always passed by reference, never by value, as they can be in C++. However, like C++ and unlike Java, Virgil has no universal super-class akin to `java.lang.Object` from which all classes ultimately inherit. But Virgil differs from C++ in two important ways; it is strongly typed, which forces explicit downcasts of object references to be checked dynamically, and it does not provide pointer types such as `void*`. This inheritance model allows for object-oriented design in applications using familiar patterns and provides for a straightforward and efficient object model implementation. Like NesC, Virgil provides a means of handling hardware interrupts in software, but it currently does not provide atomic blocks or other concurrency features; management of the interrupt state is done by the program through manipulation of hardware state exposed declaratively.

One of Virgil's key innovative features is the notion of *initialization time*, whereby an application can allocate and initialize its entire object heap during compilation using an interpreter for the complete language built into the compiler. This allows an application to build all of its data structures during compilation; memory allocation during execution on the device is disallowed. This feature is motivated by the observation that many microcontroller programs already written in C and nesC statically allocate all of their memory via global variables and reuse this memory at execution time. Static allocation removes the need for a memory management library such as `malloc()` and `free()`, or in the case of a safe language, the need for a garbage collector or other automatic memory management system. The lack of dynamic memory allocation also removes the need for programs to be made robust against the possibility of exhausting memory, since memory is statically allocated and apportioned at compile time.

The object model allows a straightforward, highly efficient implementation. In addition to its fields, the compiler may add a single-word header to each object that stores a pointer to a shared meta-object for the object's class. The meta-object contains a type identifier and a virtual dispatch table. It is read-only and stored in ROM if necessary to conserve RAM. This object header is

omitted for Virgil classes that are unrelated to any other classes (so-called *orphan* classes) because complete type information is known statically, and all virtual calls become direct calls.

Virgil elevates the common practice of static memory allocation on microcontrollers to a first-class language feature. In addition to implementing the standard phases of compilation such as parsing, typechecking, optimization, and code generation, the Virgil compiler includes an interpreter for the complete Virgil language. After typechecking is complete, the compiler executes the application’s initialization code with the built-in interpreter. The interpreter allows the application code to perform any Turing-complete computation using the complete Virgil language, including, for example, allocating objects, initializing them, calling methods on them, etc. This phase allows unrestricted allocation; a general-purpose garbage collector ensures that unreachable objects allocated by the application are discarded. When the application’s initialization routines have terminated, the compiler will generate an executable that includes not only the code of the program, but also the data of the entire reachable heap. This initial heap is immediately available to the program at runtime on the device. During execution the program can perform unrestricted computation and can read and write heap fields, but further memory allocation is disallowed and will result in an exception. We would prefer to avoid such exceptions via static program analysis; we leave that for future work.

The initialization time model provides new opportunities for compiler optimizations, since the compiler has not only the entire code of the program, but the entire initialized heap as well. This allows the compiler to employ data-sensitive optimizations that can exploit the size and shape of the heap, as well as the initial values of fields in objects. One such optimization is a combined dead code and dead data elimination optimization called *reachable members analysis*. RMA can be thought of reachability over program code and data object simultaneously; it serves to remove dead code and objects from the program that cannot be reached over any program execution. For the purposes of this work, it can be considered complementary. In fact, all of our experimental results are obtained against a baseline of performing the RMA optimization first, since it was so effective that it led to libraries that rely on RMA to remove their unused parts from the user program.

Initialization time also makes the optimizations described in this paper possible. We exploit the availability of the entire program heap at compile time to represent references specially and to layout objects in novel ways to achieve better results on small architectures.

### 3 Reference Compression

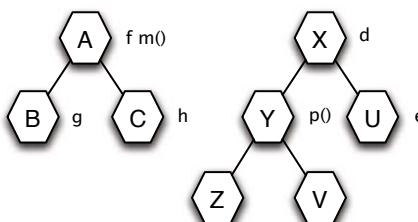
Microcontrollers have severe limitations on the RAM space available to store heap objects and the program stack, which makes a space-efficient representation of objects of primary importance. While the Virgil language was designed to have straightforward and low costs for the implementation of objects, as well as sophisticated dead code and dead data elimination passes, we would like to

employ advanced object layout strategies and compression techniques to further improve on the basic implementation. Compression has two advantages: first, it allows larger applications to be built and deployed on the same hardware; and second, it allows a given application to be deployed on a smaller, cheaper microcontroller model with less RAM.

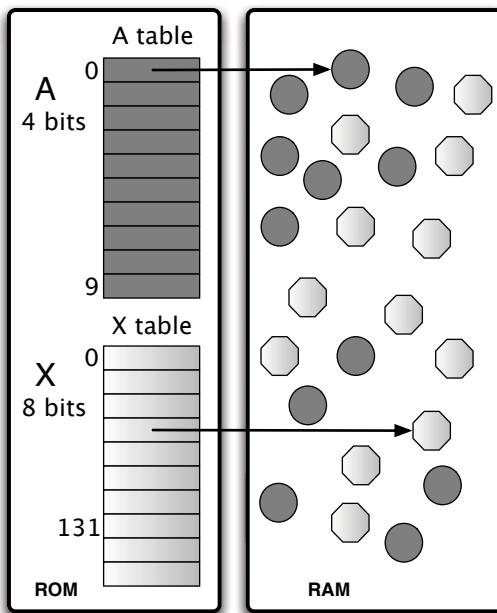
Our previous work on Virgil [20] explored table-based compression of object references for Virgil, but did not evaluate the impact on performance. This paper provides those experimental results and compares them with our new technique, compressed vertical object layout. We briefly describe the previous table-based compression technique in this section for completeness.

Reference compression exploits the strong type safety of Virgil to achieve a more compact representation of object references. In a typical high-performance object system, references are usually represented as direct byte addresses within the address space of RAM. However, direct byte addresses are not the most compact representation. For example, in a typical microcontroller system, these addresses are typically 8 or 16 bits wide, even though physical memory often does not completely fill the address space (e.g. an Atmel ATMega128 has 4KB of RAM and uses 16 bit pointers). Further, in a single address-space architecture, some of the address space is occupied by the (usually read-only) code memory. Direct byte addresses are useful for weakly typed languages like C, where a pointer is not constrained to point to values of any particular type and can conceivably hold any value. In fact, pointer arithmetic relies on the fact that pointers are represented as integers and allows operations such as increment, addition, subtraction, and conversion between types. Worse, C allows pointers to be converted to integers, manipulated, and converted back to pointers. But in Virgil, the type of a reference restricts the set of possible objects that it may reference to only those objects that are of the corresponding type or one of its subtypes. Recall that after initialization time in the compiler, a Virgil application has allocated all of its objects that will exist over any execution. The compiler can use this fact to represent references specially.

The most straightforward way to implement reference compression is to use a compression table where each compressed reference is an object handle: an integer index into a table that contains the actual addresses of each object. Because Virgil has disjoint inheritance hierarchies, the compiler can compress



**Fig. 1.** Example Virgil class hierarchy. Hexagons represent classes; their field and method declarations are to their right.



**Fig. 2.** Table-based reference compression for the example. Each root class (classes A and X) has its own compression table in ROM that stores the RAM addresses of the objects.

each reference by creating a compression table for its associated root class, with one entry in the table for each object whose type is a subclass of that root. The number of bits needed to represent the integer index is therefore the logarithm of the table size. For example, if the table has 9 live objects plus null, we could use a 4-bit integer index, a saving of 75% over storing a 16-bit address. Because there is no garbage collector which may move objects at runtime, object addresses are fixed, which allows the compiler to store the table in ROM or flash, which is considerably larger than RAM, though usually slightly slower to access. Figure 1 introduces an example class hierarchy and Figure 2 gives the corresponding table-based compression implementation.

The table adds a level of indirection to all object operations. Reads and writes on object fields require first looking up the object's RAM address from the compression table before performing the operation as before. In some situations, the compiler may be able to avoid the cost of the indirection by using standard optimizations to cache the actual address of frequently accessed objects, e.g. within loops. Accesses to reference fields in the heap may also be slower if the fields are bit-packed in memory and require masks and shifts. (However, if fields are packed only at the byte level, accesses can be faster if the field requires only one byte of storage instead of two.) Thus, table-based compression represents a classic space/time tradeoff: it consumes some ROM space for the tables and may reduce performance, but saves RAM.

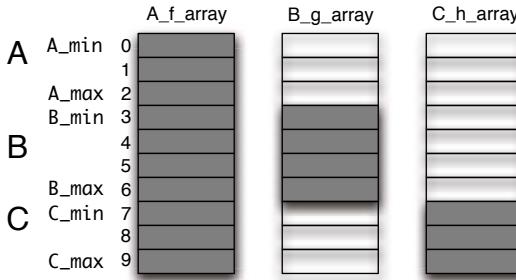
It is important to note that table-based compression can sometimes save RAM space even if the compression tables themselves are also stored in RAM. This is because for a table of size  $K$  and a pointer size of  $P$  bits, the cost of the table is  $K \times P$  bits while the savings is  $N \times (P - \log(K + 1))$  bits, where  $N$  is the number of references compressed. Note that  $N$  is always larger than  $K$  because every object must have at least one reference to it to be considered live. If  $N$  is large enough,  $N \times (P - \log(K + 1))$  is larger than  $K \times P$ . We don't expect this case to be common; our implementation always stores compression tables in ROM for maximum RAM savings.

## 4 Vertical Object Model

In traditional high-performance object-oriented systems, each object is represented in memory as a contiguous region of words that contain the values for each of the objects fields. An object reference is represented as a single-word pointer to this contiguous memory region, and the different fields of a single object are located at fixed offsets from this base address. Advanced features such as mix-ins, multiple inheritance, etc may be implemented by indirection to further contiguous memory blocks. This layout strategy has the best performance in a scenario where objects are created, moved, or reclaimed dynamically. An object allocation operation amounts to little more than an acquisition of a small contiguous region of memory, often simply bumping a top-of-heap pointer by a fixed amount. Field accesses in this model are implemented straightforwardly as a read or write of a memory address that is a small fixed offset from the object pointer; nearly all architectures allow this operation to be implemented with a single instruction. We will refer to this implementation strategy as the standard or *horizontal* layout, for reasons that will become obvious in this section.

In Virgil, the compiler has maximum freedom to layout objects in any way that respects the program's semantics because the memory layout is not exposed to the program. Our basic insight is that Virgil's initialization time model gives rise to a scenario where objects are not created, moved, or reclaimed dynamically; this means that objects need not be laid out as contiguous regions of memory words in order to simplify these operations or to allow the program to perform pointer arithmetic.

Imagine the heap of the program after initialization has completed. The program has allocated some number of objects of various types, with each object having values for all of its declared and inherited fields. If we consider this set of objects to be a two-dimensional matrix, we can consider the storage for the field's values to be entries in the matrix. Each object corresponds to a row in the matrix, and each declared field in the program corresponds to a column. If we represent objects in the standard layout, a reference corresponds to a pointer to a row of the matrix, where the elements of a single row are adjacent in memory. In a sense, the matrix is laid out *horizontally*. But one can also explore the implications of arranging this matrix in memory *vertically*, where an entire column has its elements adjacent in memory.



**Fig. 3.** Vertical object layout for the example (only classes A, B, C shown). Each field is represented by an array and only the occupied portion is stored. Preorder numbering gives objects of type A ids 0–2, B 3–6, and C 7–9.

	Horizontal	Horizontal Reference Compressed	Vertical
e instanceof A	e != null	e != -1	e != -1
e instanceof B	e != null && e->meta->id == B_metaid	e != -1 && Atable[e]->meta->id == B_metaid	e >= B_min && e <= B_max
e instanceof C	e != null && e->meta->id == C_metaid	e != -1 && Atable[e]->meta->id == C_metaid	e >= C_min
e.f	e->f	Atable[e]->f	A_f_array[e]
e.g	e->g	Atable[e]->g	B_g_array[e - B_min]
e.m()	e->meta->m(e)	Atable[e]->meta->m(e)	A_m[A_metaid[e]](e)

**Fig. 4.** Object operations from the Figure 1 example and each model's corresponding implementation (in pseudo-C). Bold expressions are constants inlined into the code by the compiler. The **Atable** is stored in ROM.

Consider again the example in Figure 1 and the corresponding vertical layout in Figure 3, along with the code snippets in Figure 4. The classes A, B and C have declared fields f, g, and h, respectively. Suppose now that we collect all the objects in the initialized heap of these types and number them so that all the objects of exact type A are first, B second, and C third. Then if we put these objects into a table such that the columns are the fields f, g, and h, we can see that each column has a contiguous range of indices for which the field is valid corresponding to the indices of the class in which the field was declared. If we represent an object reference as an index from 0 to 9 (with -1 representing a null reference), and represent the field f as an array A\_f\_array, we can read and write the field by simply indexing into A\_f\_array by the object number.

An access of the field g in the program requires the receiver object to be of type B; therefore we know statically that accesses of field g must use indices in the valid range for B objects. While we could represent the field g as an array over the entire index range 0 to 9, we can avoid wasting space by instead rebasing the array so that element 0 of the array corresponds to index 3, the first valid index for B. Then, an access of the field g for a type B would simply adjust by subtracting 3 from the object index before accessing the array. While this seems slower, it is equivalent to a base-0 array if the compiler constant-folds the known fixed address of the array and the subtraction adjustment; the compiler will just

```

void assignAll(Program p) {
    for (ClassInfo cl : p.getRootClasses() )
        assignIndices(0, cl);
}
int assignIndices(int min, ClassInfo cl) {
    int max = min;
    // assign the indices for objects of this type
    for (ObjectInfo o : cl.instances)
        o.index = max++;
    // recursively assign ids for all the children
    for (ClassInfo child : cl.getChildren())
        max = assignIndices(max, child);

    // remember the interval for this class
    cl.indices = new Interval(min, max);
    return max;
}

```

**Fig. 5.** Algorithm to compute object indices by pre-order traversal of inheritance tree. For each class, ClassInfo stores a list of the child classes and an interval representing the valid indices for objects of this class and subclasses. For each object, ObjectInfo stores the object id (index) assigned to the object.

use a known fixed address corresponding to where the array would have started in memory if it had been based at 0.

It is simple to generalize from the example. For any inheritance tree, we simply assign object identifiers using a pre-order tree traversal. Figure 5 gives the algorithm. We use () to indicate method calls with no arguments. The output of the algorithm is an interval of valid indices for each class and an object id for every object. By employing pre-order traversal of the inheritance tree, the final assignment guarantees that each class has a contiguous range of indices corresponding to all objects of that type or one of its subtypes. Therefore, the array that represents that field in the vertical object layout can be compacted without wasting space. This algorithm chooses to restart the object id at zero for each root class in the hierarchy, which means that an object id is unique within its inheritance hierarchy, but not necessarily globally unique.

We can use the same technique to represent meta-objects vertically as well. In Virgil, meta-objects store a type identifier that is used for dynamically checking downcasts and a dispatch table that is used for virtual dispatch. We can use the same algorithm to number the meta-objects according to the inheritance hierarchy and then represent each method slot in the dispatch table vertically. A virtual dispatch then amounts to two vertical field accesses (as opposed to two horizontal field accesses in the traditional approach). The first access is to get the type information of the object by indexing into the type information array using the object index. The retrieved value is a meta-object id that is then used to index into the appropriate virtual method array, which stores a direct pointer to the code of the appropriate method.

This numbering technique also has another advantage in that the contiguity of the object identifiers makes dynamic type tests extremely cheap, because the object identifier actually encodes all the type information needed for the cast. The algorithm assigns object identifiers so that every class has an interval of valid indices that correspond to all objects of that type. Thus, given a reference  $R$  that is represented by an object index and a cast to a class  $C$ , we can simply check that the index  $R$  is within the interval for the class  $C$ . This requires only two comparisons against two constants; no indirections and no memory loads are required. The range check automatically handles the case of a `null` reference, because `null` is represented with -1, which is outside of the range for any type. Figure 4 shows example object operations and the pseudo-C code that implements each operation in the various object representations.

Reference compression becomes trivial with vertical object layout. Because each object reference is now represented as an index that is bounded by the number of objects in its inheritance hierarchy, like table-based compression, it can be compressed to a smaller bit quantity. Thus, wherever the reference is stored in the heap (e.g. in the fields of other objects), it consumes less space. However, the field arrays may not be completely packed at the bit level. If the field is compressed to fewer than 8 bits, the indexing operation is more efficient if the field array is a byte array rather than packed at the bit-level, because memory is usually not bit-addressable. Our implementation does not compress references in the vertical layout to be smaller than a byte.

Vertical layout may also save more memory by eliminating the need to pad fields in order to align their addresses on word boundaries. In the horizontal object layout, compilers sometimes need to add padding between fields in the same object in order to align individual fields on word boundaries. This becomes unnecessary in vertical object layout; as long as each field array is aligned at the appropriate boundary for its type, each element in the array will be aligned by the simple virtue of being of uniform size. However, memory alignment is not generally an issue on microcontrollers.

## 5 Formalization

In this section our goals are to give a formal presentation of vertical object layout and to show in detail how vertical object layout relates to horizontal object layout. Our main technical tool is the notion of semantic algebra that was introduced by Mosses [13,14,15]. Our semantic algebras are heaps, including heaps that use a traditional horizontal layout, heaps that use a simple form of vertical layout, and heaps that use a more realistic version of vertical layout. All those heaps have the same signature and that enables us to give the semantics of an example language in an entirely parameterized fashion. The semantics has a heap as its main data structure and the semantics works with any algebra with the given signature.

Our example language focuses on programs after the initialization phase. After initialization, computation consists of code that runs while accessing a heap. We will prove two theorems that show the tight relationships among the three semantics. We use the straightforward notion of heap homomorphism to phrase and reason about those relationships.

## 5.1 Syntax

We will specify our semantics for the following language of expressions:

- (Expression)  $e ::= x \mid e.f \mid e_0.m(e_1, \dots, e_n) \mid (C)e \mid e_1.f = e_2; e \mid \ell$
- (ClassId)  $C ::= \text{the set of class names}$
- (FieldId)  $f ::= \text{the set of field names}$
- (MethodId)  $m ::= \text{the set of method names}$
- (Label)  $\ell ::= \text{the set of object labels}$

An expression is either a reference  $x$  to an argument, a field reference  $e.f$ , a method call  $e_0.m(e_1, \dots, e_n)$ , a typecast  $(C)e$ , a field update  $e_1.f = e_2$  followed by another expression  $e$ , or an object label  $\ell$ . An object label is the formalization of a pointer to the contents of the object.

The result of a computation is an object label unless the computation diverges. The initialization phase creates the object labels; each object label points to an object representation in a heap that we introduce in the following subsection. No additional objects can be created after the initialization phase so our expression languages contains no expression for object creation.

For simplicity, we assume for each method that the arguments are called  $x_1, \dots, x_n$  and that in the method header they are listed in that order.

We use  $\leq$  to denote the subtype relation among classes.

## 5.2 Heap Algebras

Inspired by Mosses [13,14,15], we define **HEAP** to be the category of algebras with carrier set  $H$  and with signature:

$$\begin{aligned} \text{getFieldValue} &: (H \times \text{Label} \times \text{FieldId}) \rightarrow \text{Label} \\ \text{getMethodBody} &: (H \times \text{Label} \times \text{MethodId}) \rightarrow \text{Expression} \\ \text{instanceof} &: (H \times \text{Label} \times \text{ClassId}) \rightarrow \{\text{true}, \text{false}\} \\ \text{fieldUpdate} &: (H \times \text{Label} \times \text{FieldId} \times \text{Label}) \rightarrow H \end{aligned}$$

The carrier set is the actual heap value, and the four operations in the signature operate on such heap values. For a given heap and a given object label, the operation `getFieldValue` extracts the value of a particular field, the operation `getMethodBody` extract the body of a particular method, the operation `instanceof` determines whether the object is an instance of particular class, and

the operation `fieldUpdate` updates the value of a particular field in the object to be a particular object label.

We use  $H$  to range over the objects of **HEAP** and we call each such object a *heap*. A *heap homomorphism*  $g : \text{HEAP} \rightarrow \text{HEAP}$  is a function that satisfies:

$$\begin{aligned}\text{getFieldValue}(H, \ell, f) &= \text{getFieldValue}(g(H), \ell, f) \\ \text{getMethodBody}(H, \ell, m) &= \text{getMethodBody}(g(H), \ell, m) \\ \text{instanceof}(H, \ell, C) &= \text{instanceof}(g(H), \ell, C) \\ g(\text{fieldUpdate}(H, \ell_1, f, \ell_2)) &= \text{fieldUpdate}(g(H), \ell_1, f, \ell_2)\end{aligned}$$

A *heap isomorphism* is a function which is both a heap homomorphism and a bijection. It is straightforward to see that the inverse of a heap isomorphism is a heap isomorphism.

We will later define three particular kinds of heaps that all have the signature above.

### 5.3 Parameterized Semantics

We will define a small-step operational semantics in which a state is of the form  $(H, e)$  and in which a step of computation is presented as  $(H, e) \rightarrow (H', e')$ . The heap  $H$  can be any heap that has the signature presented above. Here are the ten rules:

$$\frac{(H, e) \rightarrow (H', e')}{(H, e.f) \rightarrow (H', e'.f)} \quad (1)$$

$$(H, \ell.f) \rightarrow (H, \text{getFieldValue}(H, \ell, f)) \quad (2)$$

$$\frac{(H, e_0) \rightarrow (H', e'_0)}{(H, e_0.m(e_1, \dots, e_n)) \rightarrow (H', e'_0.m(e_1, \dots, e_n))} \quad (3)$$

$$\frac{(H, e_i) \rightarrow (H', e'_i)}{(H, \ell_0.m(\ell_1, \dots, \ell_{i-1}, e_i, \dots, e_n)) \rightarrow (H', \ell_0.m(\ell_1, \dots, \ell_{i-1}, e'_i, \dots, e_n))} \quad (4)$$

$$(H, \ell_0.m(\ell_1, \dots, \ell_n)) \rightarrow (H, (\text{getMethodBody}(H, \ell_0, m))[x_1 := \ell_1, \dots, x_n := \ell_n]) \quad (5)$$

$$\frac{(H, e) \rightarrow (H', e')}{(H, (C)e) \rightarrow (H', (C)e')} \quad (6)$$

$$(H, (C)\ell) \rightarrow (H, \ell) \quad (\text{if } \text{instanceof}(H, \ell, C)) \quad (7)$$

$$\frac{(H, e_1) \rightarrow (H', e'_1)}{(H, e_1.f = e_2; e) \rightarrow (H', e'_1.f = e_2; e)} \quad (8)$$

$$\frac{(H, e_2) \rightarrow (H', e'_2)}{(H, \ell_1.f = e_2; e) \rightarrow (H', \ell_1.f = e'_2; e)} \quad (9)$$

$$(H, \ell_1.f = \ell_2; e) \rightarrow (\text{fieldUpdate}(H, \ell_1, f, \ell_2), e) \quad (10)$$

The initialization phase produces the initial state for the semantics above.

The rules are straightforward: six of the rules are congruence rules, and four of the rules delegate the work to the operations on heaps. Rule (2) says that we can evaluate a field reference expression by letting the operation `getFieldValue` do the work. Rule (5) says that we can evaluate a method call by first letting the operation `getMethodBody` get the method body and then replace each formal argument  $x_i$  with an actual argument  $\ell_i$ . Rule (7) says that we can evaluate a typecast by using the operation `instanceof` to check whether the typecast should indeed proceed or get stuck. Rule (10) says that we can evaluate a field update by first letting the operation `fieldUpdate` do the work and then proceeding with evaluating the expression that follows the field update.

Theorem 1 below states the tight relationship between homomorphic heaps: if two heaps are homomorphic and each takes a step of computation, then the results are homomorphic heaps. Theorem 1 is our basis for relating semantics based on different kinds of heaps.

**Theorem 1.** *If  $g$  is a heap homomorphism and  $(H, e) \rightarrow (H', e')$ , then  $(g(H), e) \rightarrow (g(H'), e')$ .*

*Proof.* We proceed by induction on the structure of the derivation of  $(H, e) \rightarrow (H', e')$ . We have ten cases depending on the last rule used to derive  $(H, e) \rightarrow (H', e')$ .

First the case of Rule (2), that is,  $(H, \ell.f) \rightarrow (H, \text{getFieldValue}(H, \ell, f))$ . From Rule 2 we have  $(g(H), \ell.f) \rightarrow (g(H), \text{getFieldValue}(g(H), \ell, f))$ . From that  $v$  is a heap homomorphism we have:

$$(g(H), \text{getFieldValue}(g(H), \ell, f)) = (g(H), \text{getFieldValue}(H, \ell, f))$$

Second, the case of Rule (5), that is,

$$(H, \ell_0.m(\ell_1, \dots, \ell_n)) \rightarrow (H, (\text{getMethodBody}(H, \ell_0, m))[x_1 := \ell_1, \dots, x_n := \ell_n]).$$

From Rule (5) we have

$$\begin{aligned} (g(H), \ell_0.m(\ell_1, \dots, \ell_n)) &\rightarrow \\ (g(H), (\text{getMethodBody}(g(H), \ell_0, m))[x_1 := \ell_1, \dots, x_n := \ell_n]) & \end{aligned}$$

From that  $v$  is a heap homomorphism we have:

$$\begin{aligned} (g(H), (\text{getMethodBody}(g(H), \ell_0, m))[x_1 := \ell_1, \dots, x_n := \ell_n]) &= \\ (g(H), (\text{getMethodBody}(H, \ell_0, m))[x_1 := \ell_1, \dots, x_n := \ell_n]) & \end{aligned}$$

Third, the case of Rule 7, that is,  $(H, (C)\ell) \rightarrow (H, \ell)$ , if  $\text{instanceof}(H, \ell, C)$ . From Rule 7 we have

$$(g(H), (C)\ell) \rightarrow (g(H), \ell), \text{ if } \text{instanceof}(g(H), \ell, C).$$

From that  $v$  is a heap homomorphism we have:

$$\text{instanceof}(g(H), \ell, C) = \text{instanceof}(H, \ell, C).$$

Fourth, the case of Rule 10, that is,

$$(H, \ell_1.f = \ell_2; e) \rightarrow (\text{fieldUpdate}(H, \ell_1, f, \ell_2), e). \text{ From Rule 10 we have}$$

$$(g(H), \ell_1.f = \ell_2; e) \rightarrow (\text{fieldUpdate}(g(H), \ell_1, f, \ell_2), e). \text{ From that } v \text{ is a heap homomorphism we have: } \text{fieldUpdate}(g(H), \ell_1, f, \ell_2) = g(\text{fieldUpdate}(H, \ell_1, f, \ell_2)).$$

The cases of Rules 1, 3, 4, 6, 8, 9 are immediate from the induction hypothesis.  $\square$

#### 5.4 First Instantiation: Horizontal Object Layout

We now define a set of heaps that each uses horizontal object layout. For a set  $S$ , we use  $\mathcal{P}(S)$  to denote the powerset of  $S$ .

$$\begin{aligned} H \in \text{HorizontalHeap} &= (\text{Label} \rightarrow \text{FieldTable}) \cup \\ &\quad (\text{ClassId} \rightarrow \text{HorizontalMethodTable}) \cup \\ &\quad (\{\leq\} \rightarrow \mathcal{P}(\text{ClassId} \times \text{ClassId})) \\ \text{FieldTable} &= (\{\text{meta}\} \rightarrow \text{ClassId}) \cup \\ &\quad (\text{FieldId} \rightarrow \text{Label}) \\ \text{HorizontalMethodTable} &= \text{MethodId} \rightarrow \text{Expression} \end{aligned}$$

We assume that `Label`, `ClassId`, and  $\{\leq\}$  are pairwise disjoint, and that `meta`  $\notin$  `FieldId`.

Notice that each  $H \in \text{HorizontalHeap}$  has a `FieldTable` that contains the `ClassId` and the fields, and a `HorizontalMethodTable` that maps each `MethodId` to an `Expression`. Additionally, each  $H \in \text{HorizontalHeap}$  has a `FieldTable` and a representation of the subtype relation  $\leq$ . A traditional representation would represent the subtype relation via parent pointers; instead we represent the transitive closure of the parent relation.

In the definitions below, we will often use notation for function application that we have borrowed from functional programming; for example,  $(H \ell f)$  means  $(H(\ell))(f)$ . We will also use notation for defining a function in terms of an existing function  $H$ :

$$(H[\ell' \mapsto T])(\ell) = \begin{cases} T & \text{if } \ell' = \ell \\ H(\ell) & \text{otherwise} \end{cases}$$

Notice that, in the above definition, if  $\ell$  is in the domain of  $H$ , then the function  $H[\ell' \mapsto T]$  has the same domain as  $H$ , and otherwise the domain of  $H[\ell' \mapsto T]$  has one more element than the domain of  $H$ .

We can now define the four operations:

$$\begin{aligned} \text{getFieldValue}(H, \ell, f) &= H[\ell \mapsto f] \\ \text{getMethodBody}(H, \ell, m) &= H[(H[\ell \mapsto \text{meta}]) \cdot m] \\ \text{instanceof}(H, \ell, C) &= ((H[\ell \mapsto \text{meta}]), C) \in (H \leq) \\ \text{fieldUpdate}(H, \ell_1, f, \ell_2) &= H[\ell_1 \mapsto (H[\ell_1 \mapsto f])[\ell_2 \mapsto \ell_2]] \end{aligned}$$

## 5.5 Second Instantiation: Vertical Object Layout

We now define a set of heaps that each uses vertical object layout.

$$\begin{aligned} V \in \text{VerticalHeap} &= (\text{FieldId} \rightarrow \text{ObjectTable}) \cup \\ &\quad (\text{MethodId} \rightarrow \text{VerticalMethodTable}) \cup \\ &\quad (\{\text{meta}\} \rightarrow \text{IdTable}) \cup \\ &\quad (\{\leq\} \rightarrow \mathcal{P}(\text{ClassId} \times \text{ClassId})) \\ \text{ObjectTable} &= \text{Label} \rightarrow \text{Label} \\ \text{VerticalMethodTable} &= \text{ClassId} \rightarrow \text{Expression} \\ \text{IdTable} &= \text{Label} \rightarrow \text{ClassId} \end{aligned}$$

We assume that `FieldId`, `MethodId`, `{meta}`, and `{≤}` are pairwise disjoint.

Notice that fields are accessed in a `VerticalHeap` by first supplying a `FieldId` and then `Label`, rather than the other way around. Similarly, methods are accessed by first supplying a `MethodId` and then a `ClassId` rather than the other way around.

We can now define the four operations:

$$\begin{aligned} \text{getFieldValue}(V, \ell, f) &= V[f \mapsto \ell] \\ \text{getMethodBody}(V, \ell, m) &= V[m \mapsto (V[\ell \mapsto \text{meta}])] \\ \text{instanceof}(V, \ell, C) &= ((V[\ell \mapsto \text{meta}]), C) \in (V \leq) \\ \text{fieldUpdate}(V, \ell_1, f, \ell_2) &= V[f \mapsto (V(f))[\ell_1 \mapsto \ell_2]] \end{aligned}$$

The isomorphism between `HorizontalHeap` and `VerticalHeap` (Theorem 2) is straightforward. We use `VerticalHeap` as a stepping stone towards an optimized version of vertical object layout that we define next.

## 5.6 Third Instantiation: Optimized Vertical Object Layout

We now define a set of heaps that each uses optimized vertical object layout. While more realistic than the second instantiation, we leave as an exercise for the reader some further optimizations that were discussed in the previous section and some of which will be mentioned below. We will use `Nat` to denote the natural numbers  $\{0, 1, 2, \dots\}$ .

$$\begin{aligned}
O \in \text{OptVerticalHeap} = & (\text{FieldId} \rightarrow \text{ObjectTable}) \cup \\
& (\text{MethodId} \rightarrow \text{VerticalMethodTable}) \cup \\
& (\{\text{meta}\} \rightarrow \text{IdTable}) \cup \\
& (\{\text{LabelRep}\} \rightarrow (\text{Label} \rightarrow \text{Nat})) \cup \\
& (\{\text{ClassIdRep}\} \rightarrow (\text{ClassId} \rightarrow (\text{Nat} \times \text{Nat}))) \\
\text{ObjectTable} = & \text{Label} \rightarrow \text{Label} \\
\text{VerticalMethodTable} = & \text{ClassId} \rightarrow \text{Expression} \\
\text{IdTable} = & \text{Label} \rightarrow \text{ClassId}
\end{aligned}$$

In the above definition,  $(\text{Nat} \times \text{Nat})$  represents an interval. We assume that  $\text{FieldId}$ ,  $\text{MethodId}$ ,  $\{\text{meta}\}$ ,  $\{\text{LabelRep}\}$ , and  $\{\text{ClassIdRep}\}$  are pairwise disjoint. We require that:

If  $O \in \text{OptVerticalHeap}$  and  $\ell \in \text{Label}$  and  $n = (O \text{ LabelRep } \ell)$  and  $(n'_1, n'_2) = (O \text{ ClassIdRep } (O \text{ meta } \ell))$  and  $(n_1, n_2) = (O \text{ ClassIdRep } C)$ , then  $(n_1 \leq n) \wedge (n \leq n_2)$  if and only if  $(n_1 \leq n'_1) \wedge (n'_2 \leq n_2)$ . (11)

Notice that the definition of  $\text{OptVerticalHeap}$  uses the sets  $\text{ObjectTable}$ ,  $\text{VerticalMethodTable}$ , and  $\text{IdTable}$  that we also used to define  $\text{VerticalHeap}$ . The difference between a  $\text{VerticalHeap}$  and a  $\text{OptVerticalHeap}$  is that in place of the subtype relation, an  $\text{OptVerticalHeap}$  has mappings that represent **Labels** as natural numbers and represent **ClassIds** as pairs of natural numbers. That way we can model the implementation of `instanceof` as two comparisons of natural numbers.

Requirement 11 says that if  $n$  is the number that represents a particular object,  $(n'_1, n'_2)$  is the pair that represents the class of that same object, and  $(n_1, n_2)$  represents some other class, then  $(n_1 \leq n) \wedge (n \leq n_2)$  if and only if  $(n_1 \leq n'_1) \wedge (n'_2 \leq n_2)$ . In other words, an object is in the range of a class if and only if the object's own class has a range that is contained in the range of the other class. This formalizes how vertical object layout represents the subtyping relation using pairs of numbers.

We can now define the four operations:

$$\begin{aligned}
\text{getFieldValue}(O, \ell, f) &= O \ f \ \ell \\
\text{getMethodBody}(O, \ell, m) &= O \ m \ (O \text{ meta } \ell) \\
\text{instanceof}(O, \ell, C) &= \text{let } (n_1, n_2) \text{ be } (O \text{ ClassIdRep } C) \text{ in} \\
&\quad \text{let } n \text{ be } (O \text{ LabelRep } \ell) \text{ in} \\
&\quad (n_1 \leq n) \wedge (n \leq n_2) \\
\text{fieldUpdate}(O, \ell_1, f, \ell_2) &= O[f \mapsto (O(f))[\ell_1 \mapsto \ell_2]]
\end{aligned}$$

Comparisons of the form  $(n_1 \leq n) \wedge (n \leq n_2)$  will occur frequently in the remainder of this section. We leave as an exercise to the reader to define an algebra of intervals together with suitable operations such as subset, and then use that algebra in a revised version of optimized vertical object layout.

## 5.7 Relationships

Let us define a pair of functions  $v, h$  that relate `HorizontalHeap` and `VerticalHeap`:

$$\begin{aligned}
 v : \text{HorizontalHeap} &\rightarrow \text{VerticalHeap} \\
 v(H) = & (\lambda f \in \text{FieldId}. \lambda \ell \in \text{Label}. (H \ell f)) \cup \\
 & (\lambda m \in \text{MethodId}. \lambda C \in \text{ClassId}. (H C m)) \cup \\
 & (\lambda \text{meta} \in \{\text{meta}\}. \lambda \ell \in \text{Label}. (H \ell \text{meta})) \cup \\
 & (\lambda \leq .(H \leq)) \\
 h : \text{VerticalHeap} &\rightarrow \text{HorizontalHeap} \\
 h(V) = & (\lambda \ell \in \text{Label}. \\
 & (\lambda \text{meta} \in \{\text{meta}\}. (V \text{ meta } \ell)) \cup \\
 & (\lambda f \in \text{FieldId}. (V f \ell))) \cup \\
 & (\lambda C \in \text{ClassId}. \lambda m \in \text{MethodId}. (V m C)) \cup \\
 & (\lambda \leq .(V \leq))
 \end{aligned}$$

Next we introduce two functions:

$$\begin{aligned}
 \text{order} : (\text{ClassId} \rightarrow (\text{Nat} \times \text{Nat})) &\rightarrow \mathcal{P}(\text{ClassId} \times \text{ClassId}) \\
 \text{numbering} : (\mathcal{P}(\text{ClassId} \times \text{ClassId}) \times (\text{Label} \rightarrow \text{ClassId})) &\rightarrow \\
 ((\text{Label} \rightarrow \text{Nat}) \times (\text{ClassId} \rightarrow (\text{Nat} \times \text{Nat})))
 \end{aligned}$$

The `order` function abstracts from a class representation that uses pairs of numbers to a partial order:

$$\begin{aligned}
 \text{order}(G) = & \\
 \{(D, C) \mid G(D) = (n'_1, n'_2) \wedge G(C) = (n_1, n_2) \wedge (n_1 \leq n'_1) \wedge (n'_2 \leq n_2)\}
 \end{aligned}$$

The `order` function defines that  $D$  is a subtype of  $C$  if  $D$ 's interval is contained in  $C$ 's interval. We use a simpler `order` function than strictly necessary: our `order` function works only if  $G$  is injective. If  $G$  is not injective, then we may have  $G(D) = G(C)$  in which we have both  $(D, C) \in \text{order}(G)$  and  $(C, D) \in \text{order}(G)$ , which only makes sense if  $C = D$ . However, one can define a more complicated `order` function that allows  $G(D) = G(C)$  for cases where  $D \neq C$ . That will be helpful for a case where no  $C$ -objects are created so we can use the same interval for  $D$  and  $C$ . We leave this as an exercise for the reader.

The `numbering` function represents the algorithm in Figure 5. Rather than formalizing the `numbering` function, we will simply state its two key properties. Suppose  $V \in \text{VerticalHeap}$  and  $(F, G) = \text{numbering}((V \leq), (V \text{ meta}))$  and  $(n'_1, n'_2) = G(D)$  and  $(n_1, n_2) = G(C)$ . We require:

$$(D, C) \in (V \leq) \text{ if and only if } (n_1 \leq n'_1) \wedge (n'_2 \leq n_2). \quad (12)$$

$$\begin{aligned}
 \text{if } \ell \in \text{Label} \text{ and } D = (V \text{ meta } \ell) \text{ and } n = (F \ell), \text{ then} \\
 (n_1 \leq n) \wedge (n \leq n_2) \text{ if and only if } (n_1 \leq n'_1) \wedge (n'_2 \leq n_2).
 \end{aligned} \quad (13)$$

Property 12 says that the numbering scheme must ensure that  $D$  is a subtype of  $C$  if and only if  $D$ 's interval is contained in  $C$ 's interval. Property 13 says that the numbering scheme must ensure that an object is in the range of a class if and only if the object's own class has a range that is contained in the range of the other class.

The reader may have noticed the close relationships among Requirement 11 for each `OptVerticalHeap`, the definition of `order`, and Properties (12)–(13). This is by design: as we will show below, the close relationship between those definitions are the key to proving relationships between `VerticalHeap` and `OptVerticalHeap`.

Finally, we use `order` and numbering to define a pair of functions  $\alpha, \gamma$  that relate `VerticalHeap` and `OptVerticalHeap`:

$$\alpha : \text{OptVerticalHeap} \rightarrow \text{VerticalHeap}$$

$$\begin{aligned} \alpha(O) = & (\lambda f \in \text{FieldId}. \lambda \ell \in \text{Label}. (O f \ell)) \cup \\ & (\lambda m \in \text{MethodId}. \lambda C \in \text{ClassId}. (O m C)) \cup \\ & (\lambda \text{meta} \in \{\text{meta}\}. \lambda \ell \in \text{Label}. (O \text{ meta } \ell)) \cup \\ & (\lambda \leq \in \mathcal{P}(\text{ClassId} \times \text{ClassId}). (\text{order } (O \text{ ClassIdRep}))) \end{aligned}$$

$$\gamma : \text{VerticalHeap} \rightarrow \text{OptVerticalHeap}$$

$$\begin{aligned} \gamma(V) = & \text{let } (F, G) \text{ be } \text{numbering}((V \leq), (V \text{ meta})) \text{ in} \\ & (\lambda f \in \text{FieldId}. \lambda \ell \in \text{Label}. (V f \ell)) \cup \\ & (\lambda m \in \text{MethodId}. \lambda C \in \text{ClassId}. (V m C)) \cup \\ & (\lambda \text{meta} \in \{\text{meta}\}. \lambda \ell \in \text{Label}. (V \text{ meta } \ell)) \cup \\ & (\lambda \text{LabelRep} \in \{\text{LabelRep}\}. F) \cup \\ & (\lambda \text{ClassIdRep} \in \{\text{ClassIdRep}\}. G) \end{aligned}$$

The main work done by  $\alpha$  and  $\gamma$  is to convert between `VerticalHeaps` representation of subtyping as a relation and `OptVerticalHeaps` representation of subtyping via a numbering scheme for objects and classes. The function  $\alpha$  uses `order` in an essential way, while the function  $\gamma$  uses `numbering` in an essential way.

Notice that if  $V \in \text{VerticalHeap}$ , then  $\gamma(V)$  satisfies Requirement 11 because numbering has property 13.

**Theorem 2.**  $v$  and  $h$  are heap isomorphisms and each other's inverses,  $\alpha$  and  $\gamma$  are heap homomorphisms, and  $\alpha \circ \gamma = id_{\text{VerticalHeap}}$ .

*Proof.* We first show that  $h \circ v = id_{\text{HorizontalHeap}}$ . We calculate  $h(v(H))$  in three steps by first inlining the definition of  $h$ , then inlining the definition of  $v$  and applying  $v$  to the readily available arguments, and finally noticing that we have arrived at an expression that is  $\eta$ -equivalent to  $H$ :

$$\begin{aligned}
h(v(H)) &= (\lambda \ell \in \text{Label}.( \\
&\quad (\lambda \text{meta} \in \{\text{meta}\}.(v(H) \text{meta } \ell)) \cup \\
&\quad (\lambda f \in \text{FieldId}.(v(H) f \ell))) \cup \\
&\quad (\lambda C \in \text{ClassId}. \lambda m \in \text{MethodId}.(v(H) m C)) \cup \\
&\quad (\lambda \leq .(v(H) \leq)) \\
&= (\lambda \ell \in \text{Label}.( \\
&\quad (\lambda \text{meta} \in \{\text{meta}\}.(H \ell \text{ meta})) \cup \\
&\quad (\lambda f \in \text{FieldId}.(H \ell f))) \cup \\
&\quad (\lambda C \in \text{ClassId}. \lambda m \in \text{MethodId}.(H C m)) \cup \\
&\quad (\lambda \leq .(H \leq)) \\
&= H
\end{aligned}$$

Next we show that  $v \circ h = id_{\text{VerticalHeap}}$ . We calculate  $v(h(V))$  in three steps by first inlining the definition of  $v$ , then inlining the definition of  $h$  and applying  $h$  to the readily available arguments, and finally noticing that we have arrived at an expression that is  $\eta$ -equivalent to  $V$ :

$$\begin{aligned}
v(h(V)) &= (\lambda f \in \text{FieldId}. \lambda \ell \in \text{Label}.(h(V) \ell f)) \cup \\
&\quad (\lambda m \in \text{MethodId}. \lambda C \in \text{ClassId}.(h(V) C m)) \cup \\
&\quad (\lambda \text{meta} \in \{\text{meta}\}. \lambda \ell \in \text{Label}.(h(V) \ell \text{ meta})) \cup \\
&\quad (\lambda \leq .(h(V) \leq)) \\
&= (\lambda f \in \text{FieldId}. \lambda \ell \in \text{Label}.(V f \ell)) \cup \\
&\quad (\lambda m \in \text{MethodId}. \lambda C \in \text{ClassId}.(V m C)) \cup \\
&\quad (\lambda \text{meta} \in \{\text{meta}\}. \lambda \ell \in \text{Label}.(V \text{ meta } \ell)) \cup \\
&\quad (\lambda \leq .(V \leq)) \\
&= V
\end{aligned}$$

Next we show that  $v$  is a heap homomorphism. We have four equations to check. The first three are straightforward to check:

$$\begin{aligned}
\text{getFieldValue}(H, \ell, f) &= H \ell f \\
\text{getFieldValue}(v(H), \ell, f) &= v(H) f \ell \\
&= H \ell f \\
\text{getMethodBody}(H, \ell, m) &= H (H \ell \text{ meta}) m \\
\text{getMethodBody}(v(H), \ell, m) &= v(H) m (v(H) \text{ meta } \ell) \\
&= v(H) m (H \ell \text{ meta}) \\
&= H (H \ell \text{ meta}) m \\
\text{instanceof}(H, \ell, C) &= ((H \ell \text{ meta}) \leq C) \\
\text{instanceof}(v(H), \ell, C) &= ((v(H) \text{ meta } \ell) \leq C) \\
&= ((H \ell \text{ meta}) \leq C)
\end{aligned}$$

We now turn to the fourth equation:

$$\begin{aligned}
& v(\text{fieldUpdate}(H, \ell_1, f, \ell_2)) \\
&= v(H[\ell_1 \mapsto (H(\ell_1))[f \mapsto \ell_2]]) \\
&= (\lambda f' \in \text{FieldId}. \lambda \ell. ((H[\ell_1 \mapsto (H(\ell_1))[f \mapsto \ell_2]]) \ell f')) \cup \\
&\quad (\lambda m \in \text{MethodId}. \lambda C. ((H[\ell_1 \mapsto (H(\ell_1))[f \mapsto \ell_2]]) C m)) \cup \\
&\quad (\lambda \text{meta} \in \{\text{meta}\}. \lambda \ell. ((H[\ell_1 \mapsto (H(\ell_1))[f \mapsto \ell_2]]) \ell \text{meta})) \cup \\
&\quad (\lambda \leq .(H \leq)) \\
&= (\lambda f' \in \text{FieldId}. \lambda \ell. ((H[\ell_1 \mapsto (H(\ell_1))[f \mapsto \ell_2]]) \ell f')) \cup \\
&\quad (\lambda m \in \text{MethodId}. \lambda C. (H C m)) \cup \\
&\quad (\lambda \text{meta} \in \{\text{meta}\}. \lambda \ell. (H \ell \text{meta})) \cup \\
&\quad (\lambda \leq .(H \leq))
\end{aligned}$$

$$\begin{aligned}
& \text{fieldUpdate}(v(H), \ell_1, f, \ell_2) \\
&= (v(H))[f \mapsto ((v(H))(f))[\ell_1 \mapsto \ell_2]] \\
&= (v(H))[f \mapsto (\lambda \ell. (H \ell f))[\ell_1 \mapsto \ell_2]] \\
&= ((\lambda f' \in \text{FieldId}. \lambda \ell. (H \ell f')) \cup \\
&\quad (\lambda m \in \text{MethodId}. \lambda C. (H C m)) \cup \\
&\quad (\lambda \text{meta} \in \{\text{meta}\}. \lambda \ell. (H \ell \text{meta})) \cup \\
&\quad (\lambda \leq .(H \leq))) \\
&\quad [f \mapsto (\lambda \ell. (H \ell f))[\ell_1 \mapsto \ell_2]] \\
&= ((\lambda f' \in \text{FieldId}. \lambda \ell. (H \ell f')) \\
&\quad [f \mapsto (\lambda \ell. (H \ell f))[\ell_1 \mapsto \ell_2]]) \cup \\
&\quad (\lambda m \in \text{MethodId}. \lambda C. (H C m)) \cup \\
&\quad (\lambda \text{meta} \in \{\text{meta}\}. \lambda \ell. (H \ell \text{meta})) \cup \\
&\quad (\lambda \leq .(H \leq)))
\end{aligned}$$

From the two calculations, it is straightforward to see that:

$$v(\text{fieldUpdate}(H, \ell_1, f, \ell_2)) = \text{fieldUpdate}(v(H), \ell_1, f, \ell_2),$$

It is sufficient to show:

$$\begin{aligned}
& \lambda f' \in \text{FieldId}. \lambda \ell. ((H[\ell_1 \mapsto (H(\ell_1))[f \mapsto \ell_2]]) \ell f') \\
&= (\lambda f' \in \text{FieldId}. \lambda \ell. (H \ell f'))[f \mapsto (\lambda \ell. (H \ell f))[\ell_1 \mapsto \ell_2]]
\end{aligned}$$

For the case of applying both sides to  $f'$ , where  $f' \neq f$ , we have:

$$\begin{aligned}
& (\lambda f' \in \text{FieldId}. \lambda \ell. ((H[\ell_1 \mapsto (H(\ell_1))[f \mapsto \ell_2]]) \ell f'))f' \\
&= \lambda \ell. (H \ell f') \\
&= ((\lambda f' \in \text{FieldId}. \lambda \ell. (H \ell f'))[f \mapsto (\lambda \ell. (H \ell f))[\ell_1 \mapsto \ell_2]])f'
\end{aligned}$$

For the case of applying both sides to  $f$ , we have:

$$\begin{aligned}
 & (\lambda f' \in \text{FieldId}. \lambda \ell. ((H[\ell_1 \mapsto (H(\ell_1))[f \mapsto \ell_2]]) \ell f')) f \\
 &= \lambda \ell. ((H[\ell_1 \mapsto (H(\ell_1))[f \mapsto \ell_2]]) \ell f) \\
 &= ((\lambda f' \in \text{FieldId}. \lambda \ell. (H \ell f'))[f \mapsto (\lambda \ell. (H \ell f))[\ell_1 \mapsto \ell_2]]) f \\
 &= \lambda \ell. (H \ell f)[\ell_1 \mapsto \ell_2]
 \end{aligned}$$

So, it is sufficient to show:

$$\lambda \ell. ((H[\ell_1 \mapsto (H(\ell_1))[f \mapsto \ell_2]]) \ell f) = \lambda \ell. (H \ell f)[\ell_1 \mapsto \ell_2]$$

If we apply both sides to  $\ell$ , where  $\ell \neq \ell_1$ , we get  $(H \ell f)$  in each case. If we apply both sides to  $\ell_1$ , we get  $\ell_2$  in each case.

So far we have shown that  $v$  is a heap isomorphism and that  $h$  is the inverse of  $v$ . It follows that  $h$  is also a heap isomorphism.

Next we show the following three properties:

For  $V \in \text{VerticalHeap}$ :

$$(V \leq) = \text{let } (F, G) \text{ be numbering}((V \leq), (V \text{ meta})) \text{ in} \quad (14)$$

$$\text{order}(G)$$

For  $O \in \text{OptVerticalHeap}$ :

$$\begin{aligned}
 ((O \text{ meta } \ell), C) \in (\text{order}(O \text{ ClassIdRep})) \text{ iff} \\
 \text{let } (n_1, n_2) \text{ be } (O \text{ ClassIdRep } C) \text{ in} \quad (15) \\
 \text{let } n \text{ be } (O \text{ LabelRep } \ell) \text{ in} \\
 (n_1 \leq n) \wedge (n \leq n_2)
 \end{aligned}$$

For  $V \in \text{VerticalHeap}$ :

$$\begin{aligned}
 ((V \text{ meta } \ell), C) \in (V \leq) \text{ iff} \\
 \text{let } (F, G) \text{ be numbering}((V \leq), (V \text{ meta})) \text{ in} \quad (16) \\
 \text{let } (n_1, n_2) \text{ be } (G C) \text{ in} \\
 \text{let } n \text{ be } (F \ell) \text{ in} \\
 (n_1 \leq n) \wedge (n \leq n_2)
 \end{aligned}$$

If we combine the definition of  $\text{order}$  with Property (12), we get Property (14). If we combine Property (11) with the definition of  $\text{order}$ , we get Property (15). If we combine Properties (12)-(13), we get Property (16).

Next we show that  $\alpha \circ \gamma = id_{\text{VerticalHeap}}$ . It is straightforward to check that if

$$\text{let } (F, G) \text{ be numbering}((V \leq), (V \text{ meta})) \text{ in } \text{order}(G) = (V \leq)$$

then  $\alpha(\gamma(V)) = V$ . The required equation is Property (14).

Next we show that  $\alpha$  is a heap homomorphism. We have four equations to check. It is straightforward to see that the equations for `getFieldValue`, `getMethodBody`, and `fieldUpdate` are satisfied. For the equation for `instanceof` we have

$$\begin{aligned}
\text{instanceof}(O, \ell, C) &= \text{let } (n_1, n_2) \text{ be } (O \text{ ClassIdRep } C) \text{ in} \\
&\quad \text{let } n \text{ be } (O \text{ LabelRep } \ell) \text{ in} \\
&\quad (n_1 \leq n) \wedge (n \leq n_2) \\
\text{instanceof}(\alpha(O), \ell, C) &= ((\alpha(O) \text{ meta } \ell), C) \in (\alpha(O) \leq) \\
&= ((O \text{ meta } \ell), C) \in (\text{order}(O \text{ ClassIdRep}))
\end{aligned}$$

and the two sides are equal because of Property (15).

Finally we show that  $\gamma$  is a heap homomorphism. We have four equations to check. It is straightforward to see that the equations for `getFieldValue`, `getMethodBody`, and `fieldUpdate` are satisfied. For the equation for `instanceof` we have

$$\begin{aligned}
\text{instanceof}(V, \ell, C) &= ((V \text{ meta } \ell), C) \in (V \leq) \\
\text{instanceof}(\gamma(V), \ell, C) &= \text{let } (n_1, n_2) \text{ be } (\gamma(V) \text{ ClassIdRep } C) \text{ in} \\
&\quad \text{let } n \text{ be } (\gamma(V) \text{ LabelRep } \ell) \text{ in} \\
&\quad (n_1 \leq n) \wedge (n \leq n_2) \\
&= \text{let } (F, G) \text{ be numbering}((V \leq), (V \text{ meta})) \text{ in} \\
&\quad \text{let } (n_1, n_2) \text{ be } (G C) \text{ in} \\
&\quad \text{let } n \text{ be } (F \ell) \text{ in} \\
&\quad (n_1 \leq n) \wedge (n \leq n_2)
\end{aligned}$$

and the two sides are equal because of Property (16).  $\square$

Theorem 2 shows that `HorizontalHeap`, `VerticalHeap`, and `OptVerticalHeap` are tightly related and, via Theorem 1, give closely related semantics. Specifically, `HorizontalHeap` and `VerticalHeap` are isomorphic, and `VerticalHeap` and `OptVerticalHeap` are homomorphic with homomorphisms in both directions. So, Theorem 2 guarantees that execution based on `HorizontalHeap` will give the same result as execution based on `VerticalHeap` and also as execution based on `OptVerticalHeap`. Our translation of Virgil to a lower-level language uses a refined version of `OptVerticalHeap`; our semantics based on `OptVerticalHeap` would be a good basis for studying correctness of such a translation.

## 6 Experimental Results

In this section we evaluate the impact that reference compression and the vertical object model have on three program factors: code size, heap size, and execution time. Each of our benchmark programs, see Figure 6, is written entirely in Virgil and does not rely on external device drivers or libraries, but instead the device drivers necessary to run on the hardware are themselves implemented in Virgil and are included in these results. These applications target the Mica2 sensor network node that contains an ATMega128 AVR microcontroller (4KB of RAM, 128KB of

flash). Our Virgil compiler emits C code that is compiled to AVR machine code by `avr-gcc` version 4.0.3. Code size and heap size numbers correspond to the size of the `.text` and `.data` sections in the ELF executables emitted by `avr-gcc`, and thus correspond to the exact usage by the program when loaded onto the device. Precise performance numbers are obtained by using the program instrumentation capabilities [22] of the Avrora cycle-accurate AVR emulator.

We use 12 Virgil benchmark programs that are available as part of the driver kit we developed for the AVR microcontroller. `Blink` is a simple test of the timer driver, toggling the green LED twice per second; `LinkedList` is a simple program that creates and manipulates linked lists; `TestADC` repeatedly samples the analog to digital converter device; `TestUSART` transmits and receives data from the serial port; `TestSPI` stresses the serial peripheral interface driver; `TestRadio` initializes the CC1000 radio and sends some pre-computed packets; `MsgKernel1` is an SOS excerpt that sends messages between modules; `Fannkuch` is adapted from the Computer Language Shootout Benchmarks [5] and permutes arrays; `Decoder` is a bit pattern recognizer that uses a data structure similar to a Binary Decision Diagram; `Bubblesort` sorts arrays; `PolyTree` is a binary tree implementation that uses parametric types; and `BinaryTree` is the same tree implementation but uses boxed values.

We tested five configurations of the Virgil compiler (version vpc-b03-013) including the standard horizontal object layout; the four new configurations are normalized against the results of the standard layout to show relative increase and decrease in code size, heap size, and execution time. The three main configurations are: `hlrc`, which is the standard horizontal layout with table-based compression; `v1`, which is the vertical object layout without compression; and `vlrc`, which is the vertical layout with compression applied to object indices. The last configuration, `hlrcram`, is only shown for code size and execution time comparison; it corresponds to horizontal layout with reference compression, but instead of storing the compression tables in ROM, they are stored in RAM. This of course does not save RAM overall, but allows us to compare the cost of accessing ROM versus accessing RAM.

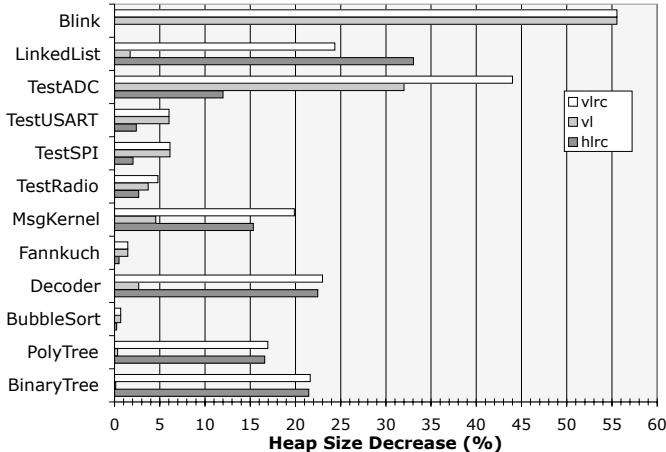
Figure 7 shows a comparison of the relative heap sizes for our benchmark programs for the three main configurations, normalized against the base configuration of horizontal layout with no reference compression. First, we notice that vertical layout (`v1`) often saves some memory over the base configuration. This is because it does not require type identifiers in the meta objects; object numbers have been assigned so that they encode the type information. Also, the horizontal layout sometimes produces zero-length objects; `avr-gcc` allocates a single byte of memory to such objects. `Blink` is a good example of that; in Figure 7, the value of `hlrc` for `Blink` is so small that the bar appears to be missing. The second observation is that the compressed vertical layout typically performs as well as the compressed horizontal layout, although some of this is due to the empty object anomaly and the lack of type identifiers in meta objects. As expected, compressed vertical layout (`vlrc`) is uniformly better than vertical layout (`v1`) alone. Note that for `Blink`, `hlrc` is so small that it

	Heap	Code	Time
BinaryTree	703	432	3,613,978
PolyTree	602	436	3,460,648
BubbleSort	874	3,878	8,419,862
Decoder	374	980	4,119,442
Fannkuch	406	5,612	951,068
MsgKernel	352	3,262	2,314,365
TestRadio	188	4,706	14,862,672
TestSPI	98	2,484	26,912
TestUSART	83	2,564	1,153,916
TestADC	50	992	53,849,859
LinkedList	115	664	1,973,094
Blink	18	778	8,038

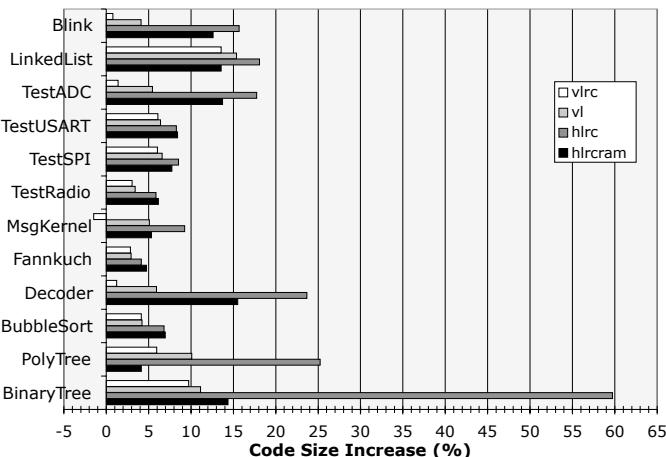
**Fig. 6.** Raw data for the standard horizontal model. Code and heap sizes are in bytes, and execution time is given in clock cycles (active cycles over 20 seconds for non-terminating programs like Blink). All other results are normalized to these.

Figure 8 shows the relative increase in code size for the same benchmarks with an added configuration, with all configurations normalized against the base configuration of horizontal layout without reference compression. Here, we can see that all configurations increase the code size of all programs (with the sole exception of `vlrc` on `MsgKernel`), with both `v1` and `vlrc` performing better than `hlrc` in each case. The increase for `vlrc` is less than 10% for most programs and less than 15% for all programs. Here, adding compression to the vertical layout actually reduces code size. This is because all field arrays become smaller, down to a single byte (because the Virgil compiler does not pack field arrays at the bit level), therefore the code to access them becomes smaller.

Horizontal reference compression increases the code size in two ways. First, it introduces compression tables that are stored in the read-only code space. Second, it requires extra instructions for each object operation due to the extra indirection. When the compression tables are stored in ROM (the `hlrc` configuration), the Virgil compiler must emit short inline AVR assembly sequences, because C does not expose the ROM address space at the source level. These assembly instructions are essentially unoptimizable by `avr-gcc`. To better isolate this effect, this figure includes code size results for a new configuration, `hlrcram` (or horizontal layout with reference compression tables in RAM). This configuration of course does not save RAM overall, but allows us to explore the effect of the special ROM assembly sequences on the code size in comparison to accessing the RAM. Comparing the `hlrc` configuration against the `hlrcram` shows that most of the code size increase is due to these special inlined ROM access sequences. The difference could be reduced if either `avr-gcc` understood and optimized accesses to ROM, or if the AVR architecture offered better addressing modes to access the ROM with fewer instructions. It is important to note that the largest proportional code size increases are for the smallest programs, as can be seen from the raw data in Figure 6.

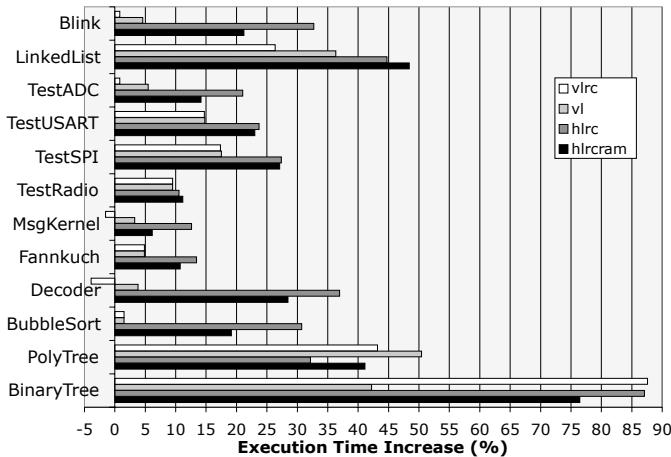


**Fig. 7.** The heap size decrease for three object models normalized against the heap size for the standard horizontal layout (larger is better)

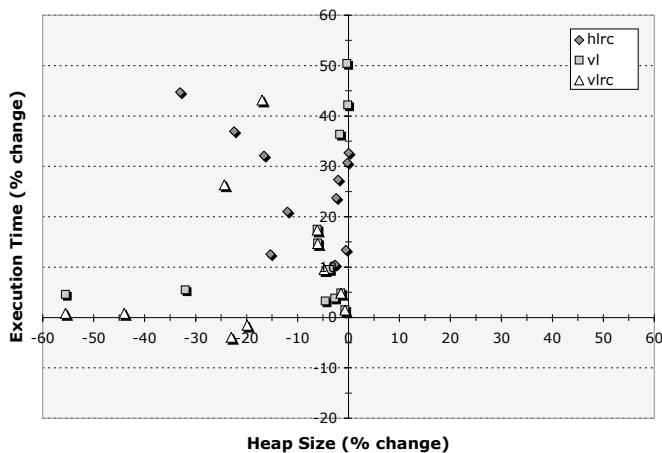


**Fig. 8.** The code size increase for four object models normalized against the code size of the standard horizontal layout (smaller is better)

Figure 9 gives the relative increase in execution time obtained by executing each benchmark in the Avrora [21] instruction-level simulator. The vertical layout technique performs better than horizontal compression in all but one case, and the execution time overhead for the compressed vertical layout is less than 20% in 9 of the 12 benchmarks, less than 10% in 7, and actually performs better than the baseline by a small amount in two cases. These two programs perform a large number of dynamic type tests, which are cheaper in the vertical layout. This figure also includes results for the `hlrcram` configuration from Figure 8 in



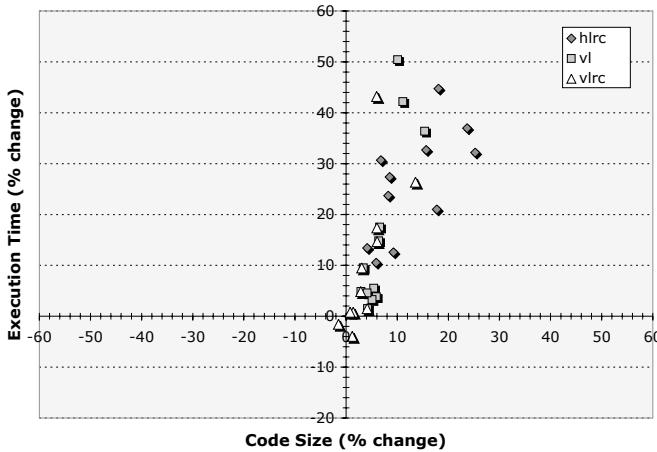
**Fig. 9.** The execution time increase for four object models normalized against the execution time of the standard horizontal layout (smaller is better)



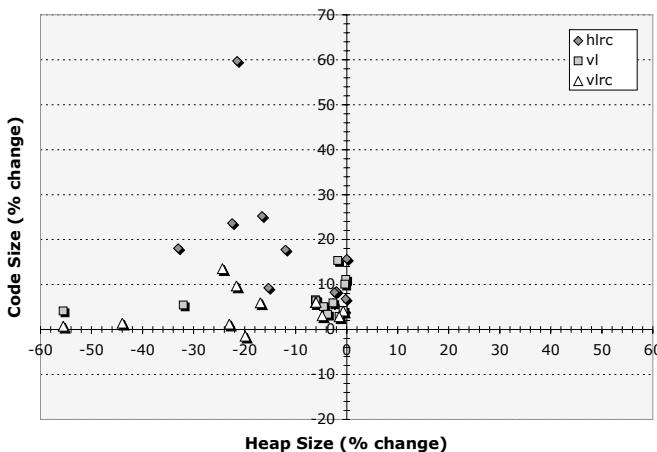
**Fig. 10.** Heap size change versus execution time change for the three object models, normalized against the standard horizontal object model

order to isolate how much of the execution time overhead is due to the cost of a ROM access versus a RAM access. In most cases, the execution time of `hlrcram` is noticeably better than that of `hlrc`, which means that a significant fraction of the overhead is due to this ROM access cost. Also notice that the largest proportional execution time increases tend to be for the smaller, pointer-intensive programs like `BinaryTree`, `PolyTree`, `LinkedList`, and `Decoder`.

Figure 10 combines the data from Figure 7 and Figure 9, showing the tradeoff between increase in execution time and the savings in heap size for the three



**Fig. 11.** Code size change versus execution time change for three object models, normalized against the standard horizontal object model



**Fig. 12.** Heap size change versus code size change for three object models, normalized against the standard horizontal object model

main configurations. First, we can see that the vertical layout without reference compression (`v1`) usually increases execution time without saving much heap space, while adding reference compression to vertical layout (`vlrc`) increases heap savings and usually has better execution time than vertical layout alone. Also, `hlrc` compression tends to have a larger increase in execution time with some savings in heap size, but not as much as `vlrc`. Overall, there is significant variation across the benchmarks, suggesting that the two factors are not intrinsically correlated. Instead, it is more likely that the factors are correlated to

benchmark characteristics, therefore the compiler should take these characteristics into account and avoid reference compression when it will save little heap space.

Figure 11 combines data from Figure 8 and Figure 9 to show the correlation between increase in code size and increase in execution time for the three main configurations. First, we can see that the two factors appear closely correlated because the points cluster near a line from the origin into the upper right quadrant. This is most likely due to the simplicity of the AVR instruction set architecture and lack of an instruction cache; adding more instructions has a predictable effect on the execution time. Second, we can see that `vlrc` performs significantly better than the other configurations, with most of its points clustered near the origin. Third, we can see that `hlrc` performs the worst, with the largest increases in code size and execution time.

Figures 12 combines the data from Figure 7 and Figure 8, comparing relative increase in code size versus decrease in heap size. Here we can see for a given heap size reduction (horizontal axis), `vlrc` tends to produce smaller code than `hlrc`.

## 7 Related Work

Cohen [8] described a constant-time subtyping test based on a “class display” where each class stores its inheritance path and subtype tests compare a class ID at a known depth. Schubert et al. [18] were the first to describe the interval encoding of type ids used here.

The Virgil notion of initialization time enables the compiler to have the entire heap available before generating code. In the traditional case where the compiler does not know all run-time objects, researchers have developed static analysis techniques that estimate a range of values for each object field. If such a range of values is small, then the compiler can optimize the heap usage by compressing fields using various strategies. For example, Ananian and Rinard [2] use static analysis of Java programs to eliminate fields with constant values and reduce the sizes of fields that can assume a small number of values. Cooprider and Regehr [9] use static analysis of C programs to pack scalars, pointers, structures, and arrays using a compression-table scheme. Lattner and Adve [11] use static analysis to convert and compress 64-bit pointers to 32 bits. Unlike these previous approaches, our compression techniques do not require computationally intensive program analysis but instead exploit the type safety of Virgil. Compilation time for all our benchmarks is less than two seconds, of which the compression time is not measurable, compared to [9] which reports analysis times measured in minutes.

While traditional static compilers do not have the complete heap, the runtime system can track all objects that have been created and use the information to dynamically compress pointers. Some research systems exist that employ dynamic techniques, sometimes assisted by hardware. For example, Zhang and Gupta [26] use special hardware instructions to help compress pointers and integers on the fly; they use profiling information to guide what data should be

compressed and when compression should be done. Chen et al. [7] use a garbage collector that compresses objects when a compacting garbage collector is not sufficient for creating space for the current allocation request; this may require dynamic decompression of objects upon their next use. Wright, Seidl, and Wolczko [25] present a memory architecture with hardware support for mapping object identifiers to physical addresses, thereby enabling new techniques for parallel and concurrent garbage collection; such an architecture could support compression of pointers as well. Wilson [24] supports large address spaces with modest word sizes by using pointer swizzling at page fault time to translate large pointers into fewer bits.

Optimization of heap usage can sometimes help performance as well. For example, Mogul et al. [12] observed in 1995 that pointer sizes could affect performance significantly on a 64-bit computer because larger pointers occupy more space, putting greater stress on the memory system, affecting cache hit ratios and paging frequency. Adl-Tabatabai et al. [1] represent 64-bit pointers as 32-bit unsigned offsets from a known base resulting in a significant performance improvement.

For object-oriented languages such as Java, each object has a header that contains such data as type information, a hash code, and a lock. Bacon, Fink, and Grove [4] presented compression techniques that allow most Java objects to have a single-word object header.

Languages such as Java and Virgil allow single inheritance of classes. For languages such as C++, List Flavors, and Theta that allow multiple inheritance among classes, researchers have developed object layouts that enable fast field access with few indirections. For example, Pugh and Weddell [17] and later Myers [16] use both positive and negative offsets of fields. It remains to be seen whether vertical object layout can be useful for languages with multiple inheritance of classes and more complex object layout models. For example, it may be possible to apply ideas from PQ-Encoding in [27].

Like most languages in common use, Virgil uses primitive types of data such as integers. Bacon [3] presented Kava, a language without primitive types in which all data is programmed in an object-oriented manner. An interesting future direction might be to explore whether our techniques can be useful for a Kava-like language with a Virgil notion of initialization time.

## 8 Conclusion and Future Work

In this paper, we evaluated static heap compression strategies that are made possible by the compilation model of Virgil, specifically, the availability of the entire program heap at compile time. Our experimental results show that programs compressed using vertical object layout have better execution time and code size than the table-based approach while achieving nearly the same RAM savings. For six of the 12 benchmark programs, vertical layout with reference compression reduces heap size by more than 20%, while no program suffers more than 15% code size increase.

Our formalization of vertical object layout validates that many aspects of our implementation are correct. However, our formalization of optimized vertical object layout contains fewer optimizations than our implementation. We leave it to future work to fully formalize all our optimizations. One can view our proof of equivalence between horizontal and vertical object layout as a step towards compiler correctness of our implementation. A standard semantics might use horizontal layout but now our equivalence theorem enables a compiler correctness proof to take its starting point in a semantics that uses vertical object layout.

The lack of dynamic memory allocation is also common in hard real-time systems and high-integrity systems. For example, SPARK [6], an industry-standard subset of Ada, disallows dynamic memory allocation in order to simplify software verification. Recently, Taha, Ellner, and Xi [19] described a functional meta-language for generating heap-bounded programs using a staged computation model and sophisticated types. The techniques described here for compression could have applicability to software written for both of these systems due to the fixed size of the heap.

Currently, the vertical object layout model requires that no new objects be created at runtime. Object allocation may require growing the field tables individually, and maintaining the contiguous nature of object identifiers might be tricky, especially in the presence of subtyping. Also, as objects become unreachable, entries in the field tables become unused and would need to be recycled. It is not clear whether the costs of such maintenance would outweigh the benefits. One might instead consider a hybrid strategy that “verticalizes” those types that are allocated only at initialization time and not at runtime. Another technique might be to hybridize both horizontal and vertical layouts for the same type in interesting ways—perhaps only part of an object is stored horizontally, and the rest of the object is stored vertically, with the index stored in the horizontal layout for access. When the class hierarchy is fixed and known statically, it is possible to layout the meta-objects (i.e. dispatch tables) vertically, even though new objects can be created at runtime. This allows the object header to be compressed to a small meta-object identifier; a virtual dispatch is then implemented as an index operation into the appropriate virtual method array.

Our compiler detects read-only component fields and object fields and inlines the values of those that are constant over all objects, but doesn’t move other read-only object fields to ROM. This would be complex in the horizontal layout model because an object might be split into a read-only portion stored in ROM and a read-write portion stored in RAM. An uncompressed horizontal object reference must point to the address of one half of the object, and that half must have a pointer to the other half. However, when compression is applied to the horizontal layout, the compiler can use one object index but instead have two compression tables, one that holds the address of the RAM portion of the object, and one that holds the ROM address of the object. Even more promising is the idea of using vertical object layout to radically simplify moving

individual fields to ROM. Because an entire field is stored contiguously and object indexes are used instead, moving a field array to ROM is trivial; the compiler can generate code to access the appropriate memory space at each field usage site. However, none of these strategies is currently implemented in the Virgil compiler.

In this work, our compiler employs a single object model for all inheritance hierarchies in the program, but one could consider a compiler that employs different object models for different hierarchies depending on the relative execution frequency of object operations and space consumption. For example, the compiler might elect to compress the most infrequently used objects using the most space efficient strategy, while employing the best performing (but possibly larger) strategy for the most frequently accessed objects. Such a compiler might employ heuristics on access frequencies or use feedback from profiling runs. We would like to extend our compiler and explore these more sophisticated strategies.

*Acknowledgments.* We thank Ryan Hall and Akop Palyan for developing many of the AVR hardware drivers in Virgil. We also thank the anonymous reviewers for many helpful suggestions. The authors were partially supported by NSF ITR award #0427202 and a research fellowship with the Center for Embedded Network Sensing at UCLA, an NSF Science and Technology Center.

## References

1. Adl-Tabatabai, A.-R., Bharadwaj, J., Cierniak, M., Eng, M., Fang, J., Lewis, B.T., Murphy, B.R., Stichnoth, J.M.: Improving 64-bit Java IPF performance by compressing heap references. In: Proceedings of CGO 2004, pp. 100–110 (2004)
2. Ananian, C.S., Rinard, M.: Data size optimizations for Java programs. In: LCTES 2003, Languages, Compilers, and Tools for Embedded Systems (2003)
3. Bacon, D.F.: Kava: a Java dialect with a uniform object model for lightweight classes. *Concurrency and Computation: Practice and Experience* 15(3–5), 185–206 (2003)
4. Bacon, D.F., Fink, S.J., Grove, D.: Space- and time-efficient implementation of the Java object model. In: Magnusson, B. (ed.) ECOOP 2002. LNCS, vol. 2374, pp. 111–132. Springer, Heidelberg (2002)
5. Bagley, D.: Computer language shootout benchmarks,  
<http://shootout.alioth.debian.org/gp4>
6. Chapman, R., Barnes, J., Dobbing, B.: On the principled design of object-oriented programming languages for high-integrity systems. In: Proceedings of the 2nd NASA/FAA Object-Oriented Technology Workshop (2003)
7. Chen, G., Kandemir, M.T., Vijaykrishnan, N., Irwin, M.J., Mathiske, B., Wolczko, M.: Heap compression for memory-constrained Java environments. In: Proceedings of OOPSLA 2003, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, pp. 282–301 (2003)

8. Cohen, N.H.: Type-extension type tests can be performed in constant time. *ACM Transactions on Programming Languages and Systems* 13(4), 626–629 (1991)
9. Cooprider, N., Regehr, J.: Offline compression for on-chip RAM. In: *Proceedings of PLDI 2007, ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, California, June 2007, pp. 363–372 (2007)
10. Gay, D., Levis, P., von Behren, J.R., Welsh, M., Brewer, E.A., Culler, D.E.: The nesC language: A holistic approach to networked embedded systems. In: *Proceedings of PLDI 2003, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 1–11 (2003)
11. Lattner, C., Adve, V.S.: Automatic pool allocation for disjoint data structures. In: *Proceedings of The Workshop on Memory Systems Performance (MSP 2002) and The International Symposium on Memory Management (ISMM 2002)*, pp. 13–24 (2002)
12. Mogul, J.C., Bartlett, J.F., Mayo, R.N., Srivastava, A.: Performance implications of multiple pointer sizes. In: *Proceedings of USENIX Winter*, pp. 187–200 (1995)
13. Mosses, P.D.: A semantic algebra for binding constructs. In: *Proceedings of ICFPC 1981, International Colloquium on Formalization of Programming Concepts*, pp. 408–418 (1981)
14. Mosses, P.D.: Abstract semantic algebras! In: *Proceedings of IFIP TC2 Working Conference on Formal Description of Programming Concepts II* (Garmisch-Partenkirchen, 1982), pp. 45–70. North-Holland, Amsterdam (1983)
15. Mosses, P.D.: A basic abstract semantic algebra. In: Plotkin, G., MacQueen, D.B., Kahn, G. (eds.) *Semantics of Data Types 1984*. LNCS, vol. 173, pp. 87–107. Springer, Heidelberg (1984)
16. Myers, A.: Bidirectional object layout for separate compilation. In: *Proceedings of OOPSLA 1995, ACM SIGPLAN Tenth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pp. 124–139 (1995)
17. Pugh, W., Weddell, G.E.: Two-directional record layout for multiple inheritance. In: *Proceedings of PLDI 1990, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 85–91 (1990)
18. Schubert, L.K., Papalaskaris, M.A., Taucher, J.: Determining type, part, color and time relationships. *IEEE Computer* 16(10), 53–60 (1983)
19. Taha, W., Ellner, S., Xi, H.: Generating heap-bounded programs in a functional setting. In: Alur, R., Lee, I. (eds.) *EMSOFT 2003*. LNCS, vol. 2855, pp. 340–355. Springer, Heidelberg (2003)
20. Titzer, B.L.: Virgil: Objects on the head of a pin. In: *Proceedings of OOPSLA 2006, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications* (2006)
21. Titzer, B.L., Lee, D.K., Palsberg, J.: Avrora: Scalable sensor network simulation with precise timing. In: *Proceedings of IPSN 2005, Fourth International Conference on Information Processing in Sensor Networks*, Los Angeles, April 2005, pp. 477–482 (2005)
22. Titzer, B.L., Palsberg, J.: Nonintrusive precision instrumentation of microcontroller software. In: *Proceedings of LCTES 2005, Conference on Languages, Compilers and Tools for Embedded Systems*, Chicago, Illinois, June 2005, pp. 59–68 (2005)
23. Titzer, B.L., Palsberg, J.: Vertical object layout and compression for fixed heaps. In: *Proceedings of CASES 2007, International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, Salzburg, Austria, September 2007, pp. 170–178 (2007)

24. Wilson, P.: Operating system support for small objects. In: Proceedings of Object Orientation in Operating Systems, pp. 80–86 (1991)
25. Wright, G., Seidl, M.L., Wolczko, M.: An object-aware memory architecture. *Science of Computer Programming* 62(2), 145–163 (2006)
26. Zhang, Y., Gupta, R.: Compressing heap data for improved memory performance. *Software – Practice & Experience* 36(10), 1081–1111 (2006)
27. Zibin, Y., Gil, J.: Efficient subtyping tests with pq-encoding. In: Proceedings of OOPSLA 2001, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, Tampa Bay, Florida, October 2001, pp. 96–107 (2001)

# Author Index

- Biernacka, Małgorzata 186  
Braga, Christiano 106  
Carvilhe, Cláudio 81  
Danvy, Olivier 162, 186  
Demangeon, Romain 250  
Doh, Kyung-Goo 274  
Gottliebsen, Hanne 297  
Guedes, Luiz Carlos Castro 227  
Haeusler, Edward Hermann 227  
Hirschkoff, Daniel 250  
Huang, Hejiao 21  
Kirchner, Hélène 21  
Klin, Bartek 121  
Krishnan, Padmanabhan 315  
Lassen, Soren B. 329
- Maidl, André Murbach 81  
Meseguer, José 43  
Musicante, Martin A. 81  
Orejas, Fernando 140  
Palsberg, Jens 1, 376  
Pari-Salas, Percy 315  
Rose, Kristoffer H. 297  
Sangiorgi, Davide 250  
Schmidt, David A. 274  
Serebrenik, A. 207  
Støvring, Kristian 329  
Titze, Ben L. 376  
van den Brand, M.G.J. 207  
van der Meer, A.P. 207  
Watt, David A. 4  
Wirsing, Martin 140