

Lecture Notes in Computer Science

788

Edited by G. Goos and J. Hartmanis

Advisory Board: W. Brauer D. Gries J. Stoer



Donald Sannella (Ed.)

Programming Languages and Systems – ESOP '94

5th European Symposium on Programming
Edinburgh, U.K., April 11-13, 1994
Proceedings

Springer-Verlag
Berlin Heidelberg New York
London Paris Tokyo
Hong Kong Barcelona
Budapest

Series Editors

Gerhard Goos
Universität Karlsruhe
Postfach 69 80
Vincenz-Priessnitz-Straße 1
D-76131 Karlsruhe, Germany

Juris Hartmanis
Cornell University
Department of Computer Science
4130 Upson Hall
Ithaca, NY 14853, USA

Volume Editor

Donald Sannella
Department of Computer Science, The University of Edinburgh
The King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, U. K.

CR Subject Classification (1991): D.3, F.3, D.2, F.4

**ISBN 3-540-57880-3 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-57880-3 Springer-Verlag New York Berlin Heidelberg**

CIP data applied for

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

© Springer-Verlag Berlin Heidelberg 1994
Printed in Germany

Typesetting: Camera-ready by author
SPIN: 10132134 45/3140-543210 - Printed on acid-free paper

Preface

This volume contains the papers selected for presentation at the 5th European Symposium on Programming (ESOP'94), which was held jointly with the 19th Colloquium on Trees in Algebra and Programming (CAAP'94) during 11–13 April in Edinburgh. ESOP is a biennial symposium which has been held previously in Saarbrücken, Nancy, Copenhagen and Rennes.

ESOP is devoted to fundamental issues in the specification, design and implementation of programming languages and systems. The emphasis is on research that bridges the gap between theory and practice, for example an implemented system that embodies an important concept or method, or a formal framework for design that usefully addresses issues arising in practical system development. The scope of the symposium includes work on: software analysis, specification, transformation, development and verification/certification; programming paradigms (functional, logic, object-oriented, concurrent, etc.) and their combinations; programming language concepts, implementation techniques and semantics; software design methodologies; typing disciplines and typechecking algorithms; and programming support tools. The programme committee received 110 submissions, from which 31 were accepted; invited papers were presented by Luca Cardelli and Robin Milner.

Programme Committee

R. Back, Turku	F. Nielson, Aarhus
G. Berry, Sophia-Antipolis	D. Sannella, Edinburgh (chairman)
G. Cousineau, Paris	A. Sernadas, Lisbon
R. De Nicola, Rome	A. Tarlecki, Warsaw
P. Dybjer, Gothenburg	D. Warren, Bristol
P. Klint, Amsterdam	R. Wilhelm, Saarbrücken
B. Krieg-Brückner, Bremen	M. Wirsing, Munich
A. Mycroft, Cambridge	

ESOP'94 was organized by the Laboratory for Foundations of Computer Science of the University of Edinburgh. It was held in cooperation with ACM SIGPLAN and was sponsored by Hewlett Packard, LFCS, and the University of Edinburgh.

Local Arrangements Committee

G. Cleland (chairman)
S. Gilmore
M. Lekuse
D. Sannella

I would like to express my sincere gratitude to the members of the programme committee and their referees (see below) for their care in reviewing and selecting the submitted papers. I am also grateful to the members of the local arrangements committee and to my secretary Tracy Combe for their hard work.

Edinburgh, January 1994

Donald Sannella

Referees for ESOP'94:

L. Aceto	P. Degano	G. Hutton	U. Montanari	H. Shi
S. Agerholm	U. de' Liguoro	P. Inverardi	L. Mounier	K. Sieber
M. Alt	R. de Simone	C.B. Jay	A. Mück	A. Sinclair
T. Altenkirch	R. DiCosmo	M. Jones	D. Murphy	G. Smolka
T. Amtoft	S. Diehl	N. Jørgensen	F. Nickl	S. Sokołowski
L. Augustsson	T. Dinesh	S. Kahrs	X. Nicollin	K. Solberg
F. Barbanera	D. Dranidis	K. Karlsson	J. Niehren	S. Soloviev
H. Barendregt	H.-D. Ehrich	A. Kennedy	H.R. Nielson	M. Srebrny
D. Basin	P. Enjalbert	N. Klarlund	A. Norman	R. Stabl
M. Bednarczyk	J. Exner	J. Klop	E.-R. Olderoog	J. Steensgaard-Madsen
I. Bengelloune	F. Fages	B. Konikowska	C. Palamidessi	
M. Benke	C. Fecht	H.-J. Kreowski	F. Parisi Presicce	B. Steffen
P. Benton	C. Ferdinand	R. Kubiak	C. Paulin	J. Stell
R. Berghammer	G. Ferrari	C. Laneve	W. Pawłowski	B. Stig Hansen
J. Berstel	G. Frandsen	M. Lara de Souza	W. Penczek	A. Stoughton
M. Białasik	T. Franzén	T. Le Sergeant	H. Peterreins	T. Streicher
A. Bockmayr	L. Fribourg	F. Lesske	B. Pierce	A. Szalas
C. Böhm	H. Ganzinger	J. Leszczyłowski	A. Piperno	D. Szczepańska
M. Boreale	P. Gardner	R. Li	K.V.S. Prasad	T. Tammet
A. Borzyszkowski	S. Gastinger	L. Liquori	C. Priami	A. Tasistro
G. Boudol	B. Gersdorf	J. Lloyd	R. Pugliese	M. Tofte
P. Bouillon	J. Gibbons	G. Longo	Z. Qian	T.H. Tse
F. Boussinot	R. Giegerich	E. MacKenzie	X. Qiwen	D. Turner
J. Bradfield	S. Gilmore	E. Madelaine	M. Raber	P. Urzyczyn
K. Bruce	S. Gnesi	C. Maeder	C. Raffalli	F. Vaandrager
M. Butler	M. Gogolla	A. Maggiolo-	W. Reif	S. van Bakel
M. Carlsson	P. Gouveia	Schettini		A. van Deursen
P. Caspi	M. Grabowski	H. Mairson	D. Rémy	J. van Wamel
G. Castagna	S. Gregory	P. Malacaria	P. Resende	E. Visser
A. Cau	G. Grudziński	K. Malmkjær	B. Reus	
M.V. Cengarle	J. Hannan	L. Mandel	M. Rittri	B. von Sydow
B. Charron-Bost	M. Hanus	D. Mandrioli	K. Rose	J. von Wright
T.-R. Chuang	T. Hardin	L. Maranget	B. Ross	P. Wadler
W. Clocksin	J. Harland	F. Maraninchì	F. Rouaix	I. Walukiewicz
M. Cole	R. Harley	E. Marchiori	J. Rutten	P. Weis
S. Conrad	P. Hartel	R. Marlet	M. Ryan	J. Winkowski
F. Corradini	J. Heering	M. Mauny	A. Sampaio	B. Wolff
V. Costa	F. Henglein	B. Mayoh	D. Sands	D. Wolz
P. Cousot	R. Hennicker	M. Mehlich	D. Sangiorgi	J. Wuertz
R. Cousot	A. Hense	A. Mifsud	O. Schoett	R. Yang
P. Cregut	C. Hermida	F. Mignard	M. Schwartzbach	W. Yi
R. Crole	D. Hofbauer	K. Mitchell	H. Seidl	M. Zawadowski
M. Danelutto	B. Hoffmann	T. Mogensen	K. Sere	W. Zhang
O. Danvy	C. Holst	B. Monsuez	C. Sernadas	S. Zhou
			K. Shen	

Contents

Invited papers

A theory of primitive objects: second-order systems <i>M. Abadi and L. Cardelli</i>	1
Pi-nets: a graphical form of π -calculus <i>R. Milner</i>	26

Contributed papers

Local type reconstruction by means of symbolic fixed point iteration <i>T. Amtoft</i>	43
An asynchronous process algebra with multiple clocks <i>H.R. Andersen and M. Mendler</i>	58
Foundational issues in implementing constraint logic programming systems <i>J.H. Andrews</i>	74
Programming with behaviors in an ML framework: the syntax and semantics of LCS <i>B. Berthomieu and T. Le Sergent</i>	89
Characterizing behavioural semantics and abstractor semantics <i>M. Bidoit, R. Hennicker and M. Wirsing</i>	105
Extending pruning techniques to polymorphic second order λ -calculus <i>L. Boerio</i>	120
λ -definition of function(al)s by normal forms <i>C. Böhm, A. Piperno and S. Guerrini</i>	135
Simulation of SOS definitions with term rewriting systems <i>K.-H. Buth</i>	150
Strategies in modular system design by interface rewriting <i>S. Cicerone and F. Parisi Presicce</i>	165
Symbolic model checking and constraint logic programming: a cross-fertilization <i>M.-M. Corsini and A. Rauzy</i>	180
A logical denotational semantics for constraint logic programming <i>A. Di Pierro and C. Palamidessi</i>	195
Compilation of head and strong reduction <i>P. Fradet</i>	211
Suffix trees in the functional programming paradigm <i>R. Giegerich and S. Kurtz</i>	225

Type classes in Haskell <i>C. Hall, K. Hammond, S. Peyton Jones and P. Wadler</i>	241
Lazy type inference for the strictness analysis of lists <i>C. Hankin and D. Le Métayer</i>	257
Lazy unification with simplification <i>M. Hanus</i>	272
Polymorphic binding-time analysis <i>F. Henglein and C. Mossin</i>	287
Shapely types and shape polymorphism <i>C.B. Jay and J.R.B. Cockett</i>	302
Bottom-up grammar analysis: a functional formulation <i>J. Jeuring and D. Swierstra</i>	317
First-class polymorphism for ML <i>S. Kahrs</i>	333
Dimension types <i>A. Kennedy</i>	348
A synergistic analysis for sharing and groundness which traces linearity <i>A. King</i>	363
A π -calculus specification of Prolog <i>B.Z. Li</i>	379
A logical framework for evolution of specifications <i>W. Li</i>	394
A semantics for higher-order functors <i>D.B. MacQueen and M. Tofte</i>	409
The PCKS-machine: an abstract machine for sound evaluation of parallel functional programs with first-class continuations <i>L. Moreau</i>	424
A tiny constraint functional logic language and its continuation semantics <i>A. Mück, T. Streicher and H.C.R. Lock</i>	439
Fully abstract translations and parametric polymorphism <i>P.W. O'Hearn and J.G. Riecke</i>	454
Broadcasting with priority <i>K.V.S. Prasad</i>	469
Towards unifying partial evaluation, deforestation, supercompilation, and GPC <i>M.H. Sørensen, R. Glück and N.D. Jones</i>	485
Algebraic proofs of properties of objects <i>D. Walker</i>	501

A Theory of Primitive Objects

Second-Order Systems

Martín Abadi and Luca Cardelli

Digital Equipment Corporation, Systems Research Center

Abstract

We describe a second-order calculus of objects. The calculus supports object subsumption, method override, and the type *Self*. It is constructed as an extension of System **F** with subtyping, recursion, and first-order object types.

1. Introduction

To its founders and practitioners, object-oriented programming is a new computational paradigm distinct from ordinary procedural programming. Objects and method invocations, in their purest form, are meant to replace procedures and calls, and not simply to complement them. Is there, then, a corresponding “ λ -calculus” of objects, based on primitives other than abstraction and application?

Such a specialized calculus may seem unnecessary, since untyped objects and methods can be reduced to untyped λ -terms. However, this reduction falters when we take typing and subtyping into account. It becomes even more problematic when we consider the peculiar second-order object-oriented concept of the *Self* type.

This paper is part of an effort to identify object calculi that are as simple and fruitful as λ -calculi. Towards this goal, we have considered untyped, first-order, and second-order systems, their equational theories, and their semantics. Here we describe, in essence, an object-oriented version of the polymorphic λ -calculus.

The starting point for this paper is a first-order calculus of objects and their types, introduced in [Abadi, Cardelli 1994c]. In this calculus, an object is a collection of methods. A method is a function having a special parameter, called *self*, that denotes the same object the method belongs to. The calculus supports *object subsumption* and *method override*. Subsumption is the ability to emulate an object by means of another object that has more refined methods. Override is the operation that modifies the behavior of an object, or class, by replacing one of its methods; the other methods are *inherited*. We review this calculus in section 2.

We add standard second-order constructs to the first-order calculus. The resulting system is an extension of Girard’s System **F** [Girard, Lafont, Taylor 1989] with objects, subtyping, and recursion. Only the first-order object constructs are new. However, the interaction of second-order types, recursive types, and objects types is a prolific one.

Using all these constructs, we define an interesting new quantifier, ζ (sigma), similar to the μ binder of recursive types. This quantifier satisfies desirable subtyping properties that μ does not satisfy, and that are important in examples with objects. Using ζ and a covariance condition we formalize the notion of *Self*, which is the type of the *self* parameter.

We take advantage of Self in some challenging examples. One is a treatment of the traditional geometric points. Another is a calculator that modifies its own methods. A third one is an object-oriented version of Scott numerals, exploiting both Self and polymorphism.

Some modern object-oriented languages support Self, subsumption, and override (e.g., [Meyer 1988]). Correctness is obtained via rather subtle conditions, if at all. We explain Self by deriving its rules from those for more basic constructs. Thus, the problems of consistency are reduced, and hopefully clarified.

Our main emphasis is on sound typing rules for objects; because of this emphasis we restrict ourselves to a stateless computational model. However, our type theories should be largely applicable to imperative and concurrent variants of the model, and our equational theories reveal difficulties that carry over as well.

In the next section we review the first-order calculus. Section 3 concerns the second-order constructs, the Self quantifier, and matters of covariance and contravariance. In section 4 we combine the Self quantifier with object types to formalize the type Self, and we present examples. In section 5 we discuss the problem of overriding methods that return values of type Self. We conclude with a comparison with related work. The appendix lists all the typing and equational rules used in the body of the paper. In [Abadi, Cardelli 1994a] we give a denotational model for these rules.

2. First-Order Calculi

In this section we review the typed first-order object calculus introduced in [Abadi, Cardelli 1994c]. We also recall its limitations, which motivate the second-order systems that are the subject of this paper.

2.1 Informal Syntax and Semantics of Objects

We consider a minimal object calculus including *object formation*, *method invocation*, and *method override*. The calculus is very simple, with just four syntactic forms, and even without functions. It is patently object-oriented: it has built-in objects, methods with self, and the characteristic semantics of method invocation and override. It can express object-flavored examples in a direct way.

Syntax of the first-order ζ -calculus

$A, B ::= [l_i=B_i \ i \in 1..n]$	(l_i distinct)	types
$a, b ::=$		terms
x		variable
$[l_i=\zeta(x_i:A)b_i \ i \in 1..n]$	(l_i distinct)	object
$a.l$		field selection / method invocation
$a.l \Leftarrow \zeta(x:A)b$		field update / method override

Notation

- We use indexed notation of the form $\Phi_i \ i \in 1..n$ to denote sequences Φ_1, \dots, Φ_n .
- We use “ \triangleq ” for “equal by definition”, “ \equiv ” for “syntactically identical”, and “ $=$ ” for “provably equal” when applied to two terms.
- $[..., l':A ...]$ stands for $[...:A, l':A:...]$.
- $[..., l=b, ...]$ stands for $[..., l=\zeta(y:A)b, ...]$, for an unused y . We call $l=b$ a *field*.
- $o.l:=b$ stands for $o.l \Leftarrow \zeta(y:A)b$, for an unused y . We call it an *update* operation.

- We write $b\{x\}$ to highlight that x may occur free in b . The substitution of a term c for the free occurrences of x in b is written $b\{x \leftarrow c\}$, or $b\{c\}$ where x is clear from context.
- We identify $\zeta(x:A)b$ with $\zeta(y:A)(b\{x \leftarrow y\})$, for any y not occurring free in b .

An object is a collection of components $l_i = a_i$, for distinct labels l_i and associated methods a_i ; the order of these components does not matter. The object containing a given method is called the method's *host* object. The symbol ζ is used as a binder for the self parameter of a method; $\zeta(x:A)b$ is a method with self parameter x of type A , to be bound to the host object, and body b .

A *field* is a degenerate method that ignores its self parameter; we talk about *field selection* and *field update*. We use the terms selection and invocation and the terms update and override somewhat interchangeably.

A method invocation is written $o.l_j$, where l_j is a label of o . It equals the result of the substitution of the host object for the self parameter in the body of the method named l_j .

A method override is written $o.l_j \Leftarrow \zeta(y:A)b$. The intent is to replace the method named l_j of o with $\zeta(y:A)b$; this is a single operation that involves a construction binding y in b . A method override equals a copy of the host object where the overridden method has been replaced by the overriding one. The semantics of override is functional; an override on an object produces a modified copy of the object.

An object of type $[l_i : B_i]_{i \in 1..n}$ can be formed from a collection of n methods whose self parameters have type $[l_i : B_i]_{i \in 1..n}$ and whose bodies have types B_1, \dots, B_n . When writing $[l_i : B_i]_{i \in 1..n}$, we always assume that the l_i are distinct and that permutations do not matter. The type $[l_i : B_i]_{i \in 1..n}$ exhibits only the result types B_i , and not the types of ζ -bound variables. The types of all these variables is $[l_i : B_i]_{i \in 1..n}$. When the method named l_i is invoked, it produces a result of type B_i . A method can be overridden while preserving the type of its host object.

Self-substitution is at the core of the semantics. Because of this, it is easy to define non-terminating computations without explicit use of recursion:

$$\text{let } o \triangleq [l = \zeta(x:[])x.l] \quad \text{then } o.l = x.l\{x \leftarrow o\} \equiv o.l = \dots$$

Using recursive types, it is possible for a method to return or to modify self:

$$\begin{aligned} \text{let } A &\triangleq \mu(X)[l:X] \\ \text{let } o' &\triangleq [l = \zeta(x:A)x] \quad \text{then } o'.l = x\{x \leftarrow o'\} \equiv o' \\ \text{let } o'' &\triangleq [l = \zeta(y:A)(y.l \Leftarrow \zeta(x:A)x)] \quad \text{then } o''.l = (o''.l \Leftarrow \zeta(x:A)x) = o' \end{aligned}$$

We place particular emphasis on the ability to modify self. In object-oriented languages, it is very common for a method to modify field components of self. Generalizing, we allow methods to override other methods of self, or even themselves. This feature does not significantly complicate the problems that we address.

We do not provide an operation to extract a method from an object as a function; such an operation is incompatible with object subsumption in typed calculi. Methods are inseparable from objects and cannot be recovered as functions; this consideration inspired the use of a specialized ζ -notation instead of the familiar λ -notation for parameters.

Other choices of primitives are possible; some are discussed in [Abadi, Cardelli 1994c].

2.2 Object Typing and Subtyping

We now review the typing and subtyping rules for objects. Each rule has a number of antecedent judgments above a horizontal line and a single conclusion judgment below the line. Each judgment has the form $E \vdash \mathcal{I}$, for an environment E and an assertion \mathcal{I} depending on the judgment. An antecedent of the form " $E, E_i \vdash \mathcal{I}_i \forall i \in 1..n$ " is an abbreviation for n antecedents

“ $E, E_1 \vdash S_1 \dots E, E_n \vdash S_n$ ” if $n > 0$, and if $n = 0$ for “ $E \vdash \diamond$ ”, which means “ E is well-formed”. Instead, a rule containing “ $j \in 1..n$ ” indicates that there are n separate rules, one for each j . Environments contain typing assumptions for variables; later they will also contain type-variable declarations and subtyping assumptions.

First we give rules for proving type judgments $E \vdash B$ (“ B is a well-formed type in the environment E ”) and value judgments $E \vdash b : B$ (“ b has type B in E ”).

(Type Object) $(l_i \text{ distinct})$	$(\text{Val } x)$	(Val Object) $(\text{where } A \equiv [l_i : B_i]_{i \in 1..n})$
$E \vdash B_i \quad \forall i \in 1..n$	$E, x:A, E' \vdash \diamond$	$E, x_i:A \vdash b_i : B_i \quad \forall i \in 1..n$
$E \vdash [l_i : B_i]_{i \in 1..n}$	$E \vdash x : A$	$E \vdash [l_i = \zeta(x_i : A)b_i]_{i \in 1..n} : A$
(Val Select)	(Val Override) $(\text{where } A \equiv [l_i : B_i]_{i \in 1..n})$	
$E \vdash a : [l_i : B_i]_{i \in 1..n} \quad j \in 1..n$	$E \vdash a : A \quad E, x:A \vdash b : B_j \quad j \in 1..n$	
		$E \vdash a.l_j : B_j \quad E \vdash a.l_j = \zeta(x : A)b : A$

A characteristic of object-oriented languages is that an object can emulate another object that has fewer methods. We call this notion *subsumption*, and say that an object can *subsume* another one. We define a particular form of subsumption that is induced by a subtyping relation between object types. An object that belongs to a given object type A also belongs to any supertype B of A , and can subsume objects in B . The judgment $E \vdash A <: B$ asserts “ A is a subtype of B in environment E ”.

(Type Top)	(Sub Top)	(Sub Object) $(l_i \text{ distinct})$	(Val Subsumption)
$E \vdash \diamond$	$E \vdash A$	$E \vdash B_i \quad \forall i \in 1..n+m$	$E \vdash a : A \quad E \vdash A <: B$
$E \vdash \text{Top}$	$E \vdash A <: \text{Top}$	$E \vdash [l_i : B_i]_{i \in 1..n+m} <: [l_i : B_i]_{i \in 1..n}$	$E \vdash a : B$

For convenience, we add a constant, **Top**, a supertype of every type. The subtyping rule for objects allows a longer object type $[l_i : B_i]_{i \in 1..n+m}$ to be a subtype of a shorter object type $[l_i : B_i]_{i \in 1..n}$. Moreover, object types are *invariant* in their components: $[l_i : B_i]_{i \in 1..n+m} <: [l_i : B_i]_{i \in 1..n}$ requires $B_i \equiv B_i$ for $i \in 1..n$. This is necessary for soundness.

The full first-order calculus of objects with subtyping is called **Ob**_{1<:}; it can be described as a union of formal system fragments $\Delta_x \cup \Delta_K \cup \Delta_{Ob} \cup \Delta_{=K} \cup \Delta_{<:K} \cup \Delta_{<:Ob}$, which are listed in appendices A and C. To facilitate comparison with other first-order calculi, **Ob**_{1<:} includes constants and their sorts ($\Delta_K \cup \Delta_{<:K}$), but in this paper we mostly ignore the related issues.

2.3 Equational Theories

We associate an equational theory with **Ob**_{1<:}, and with each of the calculi we study. The judgment $E \vdash b \leftrightarrow c : A$ asserts that b and c are equal as elements of A . The equational rules for **Ob**_{1<:} are $\Delta_{=} \cup \Delta_{\neq} \cup \Delta_{=K} \cup \Delta_{=Ob} \cup \Delta_{<:K} \cup \Delta_{<:Ob}$ from appendices A and D. We give only the main rules for objects and subtyping: two rules motivated by the use of subtyping, and two evaluation rules corresponding to the semantics of selection and override.

(Eq Subsumption)
$E \vdash a \leftrightarrow a' : A \quad E \vdash A <: B$
$E \vdash a \leftrightarrow a' : B$

(Eq Sub Object) (where $A \equiv [l_i; B_i]_{i \in 1..n}$, $A' \equiv [l_i; B_i]_{i \in 1..n}, l_j; B_j]_{j \in n+1..m}]$)
$E, x_i:A \vdash b_i : B_i \quad \forall i \in 1..n \quad E, x_i:A' \vdash b_j : B_j \quad \forall j \in n+1..m$
$E \vdash [l_i = \zeta(x_i; A) b_i]_{i \in 1..n} \leftrightarrow [l_i = \zeta(x_i; A') b_i]_{i \in 1..n+m} : A$
(Eval Select) (where $A \equiv [l_i; B_i]_{i \in 1..n}$, $a \equiv [l_i = \zeta(x_i; A') b_i]_{i \in 1..n+m}$)
$E \vdash a : A \quad j \in 1..n$
$E \vdash a.l_j \leftrightarrow b_j \{x_i \leftarrow a\} : B_j$
(Eval Override) (where $A \equiv [l_i; B_i]_{i \in 1..n}$, $a \equiv [l_i = \zeta(x_i; A') b_i]_{i \in 1..n+m}$)
$E \vdash a : A \quad E, x:A \vdash b : B_j \quad j \in 1..n$
$E \vdash a.l_j = \zeta(x; A) b \leftrightarrow [l_i = \zeta(x_i; A') b_i]_{i \in 1..n+m - \{j\}}, l_j = \zeta(x; A) b : A$

According to rule (Eq Sub Object), an object can be truncated to its externally visible methods, but only if those methods do not depend on hidden methods. The truncated object would not work otherwise.

2.4 Function Types and Recursive Types

Functions (in the form of λ -terms) can be added to $\mathbf{Ob}_{1<:}$ via standard rules, obtaining a calculus called $\mathbf{FOb}_{1<:}$. As discussed in section 3.2, functions can be encoded in terms of objects and second-order constructs.

Recursive types and values can also be added via standard rules, obtaining a calculus called $\mathbf{Ob}_{1<:\mu}$. In order to add recursive types $\mu(X)A$, we give a syntactic criterion for contractiveness in the sense of [MacQueen, Plotkin, Sethi 1986]. If A is formally contractive in the variable X , then the fixpoint $\mu(X)A$ exists and is unique. Object types are formally contractive in all their variables. The explicit isomorphism between $\mu(X)A$ and $A\{X \leftarrow \mu(X)A\}$ is given by two operators called fold and unfold.

The rules for functions and recursion are listed in appendices C and D.

2.5 The Shortcomings of First-Order Calculi

The $\mathbf{Ob}_{1<:\mu}$ calculus, consisting of objects with recursion and subtyping, is a plausible candidate as a paradigm for first-order object-oriented languages. It can be used to express many standard examples from the literature. In particular, we can write types of movable points:

$$\begin{aligned} P_1 &\triangleq \mu(X)[x:Int, mv_x:Int \rightarrow X] && \text{1-D movable points} \\ P_2 &\triangleq \mu(X)[x,y:Int, mv_x, mv_y:Int \rightarrow X] && \text{2-D movable points} \end{aligned}$$

We would then expect to obtain $P_2 <: P_1$, since intuitively P_2 extends P_1 . But this is not provable, because the invariance of object types blocks the application of the recursive subtyping rule (Sub Rec) to the result type of mv_x .

Moreover, if we somehow allow $P_2 <: P_1$, we obtain an inconsistency. Briefly, suppose we use subsumption from $p:P_2$ to $p:P_1$, and then override the mv_x method of p with one that returns a proper element of P_1 . Then, some other method of p may go wrong because it assumes that mv_x produces an element of P_2 .

Hence, the failure of $P_2 <: P_1$ is necessary. At the same time, it is unfortunate: in the common situation where a method returns an updated self, we lose all useful subsumption relations. In [Abadi, Cardelli 1994c] we discuss the standard solution used in object-oriented languages such as Simula-67 and Modula-3. This solution sacrifices static typing information that must be

recovered dynamically, thus abandoning the static typing of subsumption. This paper describes another solution that preserves static typing by taking advantage of second-order constructs.

3. Second-Order Calculi

In this section we present standard second-order extensions of first-order calculi. No new or unusual constructions are introduced. However, second-order quantifiers can be combined with recursive types to produce an interesting new concept that is recognizable as formalizing the type *Self*. The interaction of *Self* with object types is the subject of section 4.

We first introduce universal quantifiers and existential quantifiers. From existential quantifiers and recursion we define the *Self* quantifier. For the purpose of defining *Self*, only existentials and recursion are needed; one could dispense with universals. For the purposes of object-oriented languages, only the *Self* quantifier is needed; one could dispense with most of the second-order baggage. However, as usual, universal quantifiers may still be useful to provide polymorphism, and existential quantifiers to provide data abstraction.

We use the notation $B\{X\}$ to single out the occurrences of X free in B ; then $B\{A\}$ stands for $B\{X \leftarrow A\}$ when X is clear from the context.

3.1 Universal and Existential Quantifiers

Bounded universal quantifiers $\forall(X \leftarrow A)B$ are adopted with the rules of [Cardelli, *et al.* 1991; Curien, Ghelli 1992]. Bounded existential quantifiers $\exists(X \leftarrow A)B$ [Cardelli, Wegner 1985] can be encoded as $\forall(Y \leftarrow \text{Top})(\forall(X \leftarrow A)B\{X \rightarrow Y\} \rightarrow Y)$. However, this encoding does not validate a natural and desirable η rule, called (Eval Repack \leftarrow) in appendix D. Therefore, we take bounded existentials as primitive.

We assemble the following second-order calculi using fragments defined in the appendix:

$$\begin{array}{lll} \mathbf{F}_{\leftarrow} & \triangleq & \Delta_x \cup \Delta_{\rightarrow} \cup \Delta_{\leq} \cup \Delta_{\leq:X} \cup \Delta_{\leq \rightarrow} \cup \Delta_{\leq \forall} \cup \Delta_{\leq:\exists} & \mathbf{F}_{\leftarrow:\mu} & \triangleq & \mathbf{F}_{\leftarrow} \cup \Delta_{\leq:\mu} \\ \mathbf{Ob}_{\leftarrow} & \triangleq & \Delta_x \cup \Delta_{\text{Ob}} \cup \Delta_{\leq} \cup \Delta_{\leq:X} \cup \Delta_{\leq:\text{Ob}} \cup \Delta_{\leq \forall} \cup \Delta_{\leq:\exists} & \mathbf{Ob}_{\leftarrow:\mu} & \triangleq & \mathbf{Ob}_{\leftarrow} \cup \Delta_{\leq:\mu} \\ \mathbf{FOB}_{\leftarrow} & \triangleq & \mathbf{F}_{\leftarrow} \cup \Delta_{\text{Ob}} \cup \Delta_{\leq:\text{Ob}} & \mathbf{FOB}_{\leftarrow:\mu} & \triangleq & \mathbf{FOB}_{\leftarrow} \cup \Delta_{\leq:\mu} \end{array}$$

\mathbf{F}_{\leftarrow} is described in [Cardelli, *et al.* 1991], but we assume only the simpler equational theory used in [Curien, Ghelli 1992], and we add existentials. Constant types are left out because free algebras can be encoded [Böhm, Berarducci 1985]. The necessary equational fragments are left implicit, since they can be easily identified (see appendix D). For simplicity we adopt conservative equational rules for existentials, although more ambitious rules may have advantages in combination with objects.

Universals are contravariant and existentials are covariant in their bounds, as reflected by the rules (Sub All) and (Sub Exists). Moreover, if B is covariant in X (written $B\{X^+\}$), then $\exists(X \leftarrow A)B\{X^+\}$ is isomorphic to $B\{A^+\}$. This isomorphism does not necessarily hold otherwise. For example, $[l_i:B_i\{X^+\} \ i \in 1..n]$ is not covariant in X even though each $B_i\{X^+\}$ is, and so $\exists(X \leftarrow A)[l_i:B_i\{X^+\} \ i \in 1..n]$ is not isomorphic to $[l_i:B_i\{A^+\} \ i \in 1..n]$.

In the next section we show that \mathbf{Ob}_{\leftarrow} can encode \mathbf{F}_{\leftarrow} , and that $\mathbf{Ob}_{\leftarrow:\mu}$ can encode $\mathbf{F}_{\leftarrow:\mu}$ (ignoring the first-order η rule). We leave as an open problem whether \mathbf{Ob}_{\leftarrow} can be translated into a typed λ -calculus without objects while preserving subtyping. In [Abadi, Cardelli 1994a] we deal with encodings that translate subtypings into coercions.

3.2 Encodings of Product and Function types

Object types can encode product and function types in calculi without subtyping [Abadi, Cardelli 1994c], validating β -reductions. When subtyping is added, the encodings yield invari-

ant product and function types. We review the translation of function types. Strictly speaking, it is defined on type derivations, but we write it as a translation of type-annotated λ -terms.

Translation of invariant function types

$\langle\!\langle A \rightarrow B \rangle\!\rangle \triangleq [\text{arg}:\langle\!\langle A \rangle\!\rangle, \text{val}:\langle\!\langle B \rangle\!\rangle]$	
$\langle\!\langle b_{A \rightarrow B}(a_A) \rangle\!\rangle_\rho \triangleq$	$\rho \in \text{Var} \rightarrow \text{Term}$ (with $\langle\!\langle x_A \rangle\!\rangle_\rho \triangleq \rho(x)$)
$(\langle\!\langle b \rangle\!\rangle_\rho.\text{arg} \Leftarrow \zeta(x:\langle\!\langle A \rightarrow B \rangle\!\rangle) \langle\!\langle a \rangle\!\rangle_\rho).\text{val}$	for $x \notin \text{FV}(\langle\!\langle a \rangle\!\rangle_\rho)$
$\langle\!\langle \lambda(x:A)b_B \rangle\!\rangle_\rho \triangleq$	
$[\text{arg} = \zeta(x:\langle\!\langle A \rightarrow B \rangle\!\rangle) x.\text{arg},$	
$\text{val} = \zeta(x:\langle\!\langle A \rightarrow B \rangle\!\rangle) \langle\!\langle b \rangle\!\rangle_\rho \{x \leftarrow x.\text{arg}\}]$	

Similarly, product types can be defined by $\langle\!\langle A \times B \rangle\!\rangle \triangleq [\text{fst}:\langle\!\langle A \rangle\!\rangle, \text{snd}:\langle\!\langle B \rangle\!\rangle]$.

These encodings yield invariant product and function types because object types are invariant. At the second order, though, we have quantifiers that are variant in their bounds; combining them with object types, we can define a variant version of function types:

$$A \rightarrow B \triangleq \forall(X <: A) \exists(Y <: B) [\text{arg}:X, \text{val}:Y]$$

We obtain $A \rightarrow B <: A' \rightarrow B'$ if $A' <: A$ and $B <: B'$.

Translation of variant function types

$\langle\!\langle A \rightarrow B \rangle\!\rangle \triangleq \forall(X <: \langle\!\langle A \rangle\!\rangle) \exists(Y <: \langle\!\langle B \rangle\!\rangle) [\text{arg}:X, \text{val}:Y]$	
$\langle\!\langle b_{A \rightarrow B}(a_A) \rangle\!\rangle_\rho \triangleq$	$\rho \in \text{Var} \rightarrow \text{Term}$ (with $\langle\!\langle x_A \rangle\!\rangle_\rho \triangleq \rho(x)$)
open $\langle\!\langle b \rangle\!\rangle_\rho(\langle\!\langle A \rangle\!\rangle)$ as $Y <: \langle\!\langle B \rangle\!\rangle$, $y:[\text{arg}:\langle\!\langle A \rangle\!\rangle, \text{val}:Y]$	
in $(y.\text{arg} \Leftarrow \zeta(x:[\text{arg}:\langle\!\langle A \rangle\!\rangle, \text{val}:Y]) \langle\!\langle a \rangle\!\rangle_\rho).\text{val}$	for $Y, y, x \notin \text{FV}(\langle\!\langle a \rangle\!\rangle_\rho)$
$\langle\!\langle \lambda(x:A)b_B \rangle\!\rangle_\rho \triangleq$	
$\lambda(X <: \langle\!\langle A \rangle\!\rangle)$	
(pack $Y <: \langle\!\langle B \rangle\!\rangle = \langle\!\langle B \rangle\!\rangle$,	
$[\text{arg} = \zeta(x:[\text{arg}:X, \text{val}:\langle\!\langle B \rangle\!\rangle]) x.\text{arg},$	
$\text{val} = \zeta(x:[\text{arg}:X, \text{val}:\langle\!\langle B \rangle\!\rangle]) \langle\!\langle b \rangle\!\rangle_\rho \{x \leftarrow x.\text{arg}\}]$	
: $[\text{arg}:X, \text{val}:Y]$	

This idea gives rise to an encoding of the first-order λ -calculus with subtyping but no η rule into $\mathbf{Ob}_{<:}$. The translation can be extended to recursive types since $A \rightarrow B$ is contractive in any X , and trivially also to second-order quantifiers. Hence our largest calculus, $\mathbf{FOb}_{<:\mu}$, can be embedded inside $\mathbf{Ob}_{<:\mu}$. We therefore consider $\mathbf{Ob}_{<:\mu}$ as our final pure object calculus.

Trivially, we can obtain covariant product types, since these can be represented in terms of universal quantifiers and variant function types in $\mathbf{F}_{<:}$. A direct encoding is possible as well:

$$A \times B \triangleq \exists(X <: A) \exists(Y <: B) [\text{fst}:X, \text{snd}:Y]$$

We obtain $A \times B <: A' \times B'$ if $A <: A'$ and $B <: B'$.

In [Cardelli 1991] it is shown that covariant record types $\langle l_1:A_1 \dots l_n:A_n \rangle$ can be represented in $\mathbf{F}_{<:}$, using covariant product types. Hence, they are also available in $\mathbf{Ob}_{<:}$.

3.3 An Encoding of Variant Object Types

In [Abadi, Cardelli 1994c] we observe that a general covariance rule for object components is unsound. However, as in the encoding of the λ -calculus just given, we can make covariant

any component whose method is only invoked (like val), and contravariant any component whose method is only overridden (like arg).

The idea for obtaining covariance is exactly the one used for the λ -calculus. We rewrite an object type $[m:B, \dots]$ as $\exists(Y <: B) [m:Y, \dots]$. The former is invariant in B , while the latter is covariant in B . The existential quantifier still allows the invocation of m , but blocks overrides of m from the outside, since the quantifier hides the representation type Y .

The idea for obtaining contravariance is more complicated; the technique used for the λ -calculus does not seem to generalize. To make a component $m:B$ contravariant in an object type $A \triangleq [m:B, \dots]$, we first rewrite the type as $A' \triangleq \mu(X) \exists(Y <:(X \rightarrow B) \rightarrow X) [m_{in}:Y, m:B, \dots]$, where m_{in} is a new method name. Note that A' is still invariant in B . We simulate an override to the method m of an object o by writing the invocation $o.m_{in}(\lambda(s:A')b')$ instead of $o.m \Leftarrow \zeta(s:A)b$, where b' imitates b . Our intent is that an invocation of m_{in} with argument $\lambda(s:A')b'$ will override m internally. Hence the code for a typical object o of type A' will be:

$$in([m_{in} = \zeta(s:UA') \lambda(f: A' \rightarrow B) in(s.m \Leftarrow \zeta(s:UA') f(in(s))), m = \zeta(s:UA') s.m, \dots]) : A'$$

where $UA' \triangleq [m_{in}:(A' \rightarrow B) \rightarrow A', m:B, \dots]$, and for any $a:UA'$

$$in(a) : A' \triangleq \text{fold}(A', \text{pack } Y <:(A' \rightarrow B) \rightarrow A' = (A' \rightarrow B) \rightarrow A', a : [m_{in}:Y, m:B, \dots])$$

Finally we use subsumption to forget the m component, so that o has the type:

$$\mu(X) \exists(Y <:(X \rightarrow B) \rightarrow X) [m_{in}:Y, \dots]$$

which is contravariant in B . Once o has this type, its method m cannot be invoked from the outside since it is not even visible.

These techniques for obtaining covariance and contravariance are suggestive but not fully satisfactory. For example, after making two components covariant, we are no longer able to re-order them, since $\exists(X <: A) \exists(Y <: B) C$ and $\exists(Y <: B) \exists(X <: A) C$ are not equivalent types. Still, these techniques may inspire an encoding or a semantics for a language with built-in object types with covariant and contravariant components.

3.4 The Self Quantifier

Within the second-order ζ -calculus with bounded quantifiers and recursion, $\mathbf{Ob}_{\zeta:\mu}$, we can encode an interesting construction that we call the *Self quantifier*. The Self quantifier is a combination of recursion and bounded existentials, with recursion going “through the bound”:

$$\zeta(X)B \triangleq \mu(Y)\exists(X <: Y)B \quad (\text{Y not occurring in } B)$$

In general, any type $B\{A\}$ can be transformed into a type $\exists(Y <: A)B\{Y\}$ covariant in A . (Recall though that these types are not always isomorphic.) An analogous technique applies to recursive types, and motivates our definition of the Self quantifier. Given an equation $X = B\{X\}$, we transform it into $X = \exists(Y <: X)B\{Y\}$. The solution to this equation, namely $\zeta(X)B\{X\}$, satisfies the subtyping property:

$$\text{if } B\{Y\} <: B'\{Y\} \text{ then } \zeta(X)B\{Y\} <: \zeta(X)B'\{Y\},$$

even though we may not have $\mu(X)B\{X\} <: \mu(X)B'\{X\}$.

Modulo an unfolding, $\zeta(X)B$ is the same as $\exists(X <: \zeta(X)B)B$. Hence, by analogy with the standard interpretation of existential types, $\zeta(X)B\{X\}$ can be understood informally as the type of pairs $\langle C, c \rangle$ consisting of a subtype C of $\zeta(X)B\{X\}$ and an element c of $B\{C\}$.

For example, suppose we have an element x of type $\zeta(X)X$. Then, choosing $\zeta(X)X$ as the required subtype of $\zeta(X)X$, we obtain $\langle \zeta(X)X, x \rangle : \zeta(X)X$. Therefore we can construct:

$$\mu(x) \langle \zeta(X)X, x \rangle : \zeta(X)X$$

Less trivially, suppose we want to build a memory cell $m:M$ with a read operation $rd:Nat$ and a write operation $wr:Nat \rightarrow M$. We can define:

$$M \triangleq \zeta(X)[rd:Nat, wr:Nat \rightarrow X]$$

where the wr method should use its argument to override the rd field. For convenience, we adopt the following abbreviation to unfold a Self quantifier:

$$A(C) \triangleq B\{C\} \quad \text{whenever } A \equiv \zeta(X)B\{X\} \text{ and } C <: A$$

So, for example, $M(M) \equiv [rd:Nat, wr:Nat \rightarrow M]$.

To define a memory cell, we are going to use twice the fact that if $x:M(M)$, then $\langle M, x \rangle : M$. First, we need a method body for wr that, with self $s:M(M)$ and argument $n:Nat$, produces a result of type M . Since $s.rd:=n$ has the same type as s , namely $M(M)$, we can use $\langle M, s.rd:=n \rangle : M$ as the body of the wr method. Therefore, we have:

$$m: M \triangleq \langle M, [rd = 0, wr = \zeta(s:M(M)) \lambda(n:Nat) \langle M, s.rd:=n \rangle] \rangle$$

Building on these intuitions, we now study the abstract properties of the Self quantifier. The two basic operations for ζ are similar to the ones for existentials. One operation constructs an element of $\zeta(X)B$, given a subtype of $\zeta(X)B$ and an appropriate value; it is the composition of pack for existentials and fold for recursive types. The other operation inspects an element of $\zeta(X)B$ (as much as possible) and computes with its contents; it is the composition of unfold for recursive types and open for existentials.

The operation for constructing elements of type $\zeta(X)B$, in full generality, binds a type variable. Hence, we need a more complex syntax than the pairing $\langle -, - \rangle$ used above. We reuse the symbol ζ for this second-order construct, in the same way we use λ for both first-order and second-order binders. We refine the notation $\langle C, b \rangle$ to $\zeta(Y <: A = C)b$. The term $\zeta(Y <: A = C)b$ binds C to Y in b , and requires C to be a subtype of A . Within b , the types Y and C are equivalent. The type of the whole term is A .

We define, for $A \equiv \zeta(X)B\{X\}$, $C <: A$, and $b\{C\}:B\{C\}$:

$$\begin{aligned} \zeta(Y <: A = C) b\{Y\} &\triangleq \\ \text{fold}(A, (\text{pack } Y <: A = C, b\{Y\}:B\{Y\})) & \end{aligned}$$

and, for $c:A$ and $d\{Y,y\}:D$, where Y does not occur in D :

$$\begin{aligned} (\text{use } c \text{ as } Y <: A, y:B\{Y\} \text{ in } d\{Y,y\}:D) &\triangleq \\ (\text{open } \text{unfold}(c) \text{ as } Y <: A, y:B\{Y\} \text{ in } d\{Y,y\}:D) & \end{aligned}$$

The following rules for the Self quantifier can be derived from the rules for μ and \exists . Note in particular the expected subtyping rule, (Sub Self).

Δ_ζ

(Type Self)	(Sub Self)
$E, X <: \text{Top} \vdash B \quad B > X$	$E, X <: \text{Top} \vdash B <: B' \quad B, B' > X$
$E \vdash \zeta(X)B$	$E \vdash \zeta(X)B <: \zeta(X)B'$

(Val Self) (where $A \equiv \zeta(X)B\{X\}$)

$E \vdash C <: A \quad E \vdash b\{C\} : B\{C\}$

$E \vdash \zeta(Y <: A = C)b\{Y\} : A$

(Val Use) (where $A \equiv \zeta(X)B\{X\}$)

$E \vdash c : A \quad E \vdash d : D \quad E, Y <: A, y : B\{Y\} \vdash d : D$

$E \vdash (\text{use } c \text{ as } Y <: A, y : B\{Y\} \text{ in } d : D) : D$

Δ_ζ

(Eq Self) (where $A \equiv \zeta(X)B\{X\}, A' \equiv \zeta(X)B'\{X\}$)

$E \vdash C <: A' \quad E \vdash A \quad E, X <: \text{Top} \vdash B'\{X\} <: B\{X\} \quad E \vdash b\{C\} \leftrightarrow b'\{C\} : B'\{C\}$

$E \vdash \zeta(Y <: A = C)b\{Y\} \leftrightarrow \zeta(Y <: A' = C)b'\{Y\} : A$

(Eq Use) (where $A \equiv \zeta(X)B\{X\}$)

$E \vdash c \leftrightarrow c' : A \quad E \vdash d : D \quad E, Y <: A, y : B\{Y\} \vdash d \leftrightarrow d' : D$

$E \vdash (\text{use } c \text{ as } Y <: A, y : B\{Y\} \text{ in } d : D) \leftrightarrow (\text{use } c' \text{ as } Y <: A, y : B\{Y\} \text{ in } d' : D) : D$

(Eval Unself) (where $A \equiv \zeta(X)B\{X\}, c \equiv \zeta(Z <: A = C)b\{Z\}$)

$E \vdash c : A \quad E \vdash D \quad E, Y <: A, y : B\{Y\} \vdash d\{Y, y\} : D$

$E \vdash (\text{use } c \text{ as } Y <: A, y : B\{Y\} \text{ in } d\{Y, y\} : D) \leftrightarrow d\{C, b\{C\}\} : D$

(Eval Reself) (where $A \equiv \zeta(X)B\{X\}$)

$E \vdash b : A \quad E, y : A \vdash d\{y\} : D$

$E \vdash (\text{use } b \text{ as } Y <: A, y : B\{Y\} \text{ in } d\{\zeta(Y <: A = Y)y\} : D) \leftrightarrow d\{b\} : D$

Notation We write:

- $\zeta(A, c)$ for $\zeta(X <: A = A)c$ when X does not occur in c
- $\zeta(X = A)c\{X\}$ for $\zeta(X <: A = A)c\{X\}$

The memory cell definition can now be understood formally, with the $\zeta(M, \dots)$ notation replacing the informal notation $\langle M, \dots \rangle$:

$$M \triangleq \zeta(\text{Self})[\text{rd:Nat}, \text{wr:Nat} \rightarrow \text{Self}]$$

$$m: M \triangleq \zeta(\text{Self}=M) [\text{rd} = 0, \text{wr} = \zeta(s:M(\text{Self})) \lambda(n:\text{Nat}) \zeta(\text{Self}, s.\text{rd}:=n)]$$

Later examples frequently adopt this choice of Self as a bound variable.

3.5 A Pure Second-Order Object Calculus

We conclude this section by considering a pure second-order object calculus, ζOb , based exclusively on object types and the Self quantifier (see appendix E):

$$\zeta\text{Ob} \triangleq \Delta_x \cup \Delta_{\text{Ob}} \cup \Delta_c \cup \Delta_{<:x} \cup \Delta_{<:\text{Ob}} \cup \Delta_\zeta$$

This calculus departs from second-order λ -calculi by omitting function types and the standard quantifiers. It seems that, in ζOb , the only function types that can be encoded are invariant, and that the standard second-order quantifiers are not expressible. Still, as the next section demonstrates, the Self quantifier provides an essential second-order feature of object calculi, namely the type Self, with a form of type recursion. Interestingly, then, ζOb is a very small second-order object calculus that covers a spectrum of object-oriented notions.

4. Object Types with Self

The payoff of the Self quantifier comes when it is used in conjunction with object types. Object types with Self are obtained by the combination of the simple object types of section 2.2 with the Self quantifier of section 3.4. These new types allow subsumption between objects containing methods that return self.

In this section, we first derive the rules for the combination of objects with Self. We then show how these derived rules can easily provide typings for some interesting examples. We work entirely within $\mathbf{Ob}_{\zeta\mu}$, that is, within the second-order calculus with bounded quantifiers, recursion, and simple object types.

4.1 ζ -Objects

In this section we examine types of the form:

$$\zeta(X)[l_i; B_i\{X^+\}_{i \in 1..n}] \quad \text{where } B\{X^+\} \text{ indicates that } X \text{ occurs only covariantly in } B$$

We call these structures ζ -object types, and ζ -objects their corresponding values. The parameter X in $\zeta(X)[l_i; B_i\{X^+\}_{i \in 1..n}]$ is intended as the Self type. Note that we do not require that $[l_i; B_i\{X\}_{i \in 1..n}]$ be covariant in X , only that each $B_i\{X\}$ be covariant in X .

Although ζ -object types are obtained by applying a Self quantifier (which has no covariance restrictions) to an object type, for the most part we consider $\zeta(X)[l_i; B_i\{X^+\}_{i \in 1..n}]$ as a single type construction. The covariance requirement is necessary when selecting components of ζ -objects. For emphasis, we use a special syntax for the combination of Self quantifiers, object types, and the covariance requirement:

$$\zeta(X^+)[l_i; B_i\{X\}_{i \in 1..n}] \triangleq \zeta(X)[l_i; B_i\{X\}_{i \in 1..n}] \quad \text{when } X \text{ occurs only covariantly in the } B_i$$

The covariance requirement implies that X_i must not occur within any object type within B_i , since object types are invariant in their components. For example, $\zeta(X)[l_i; \zeta(Y)[m:X, n:Y]]$ violates the covariance requirement. Hence, informally, we may say that Self types do not nest: there is a single meaningful Self type within each pair of object brackets.

It is often useful to consider an unfolding $A(C)$ of a ζ -object type A :

$$A(C) \triangleq [l_i; B_i\{C\}_{i \in 1..n}] \quad \text{whenever } A \equiv \zeta(X^+)[l_i; B_i\{X\}_{i \in 1..n}] \text{ and } C <: A$$

We frequently consider $A(X)$, for a variable X , and the *self-unfolding* $A(A)$ of A . (When building an element of type A it is very common to build first an element of type $A(A)$.) We say that a type $C <: A$ and an element of $A(C)$ constitute an *implementation* of A , since they can be used to build an element of A . Then C is the *representation type* for the implementation.

The operations on ζ -objects are defined as follows. Assume that a has type A with $A \equiv \zeta(X^+)[l_i; B_i\{X\}_{i \in 1..n}]$ and $A(X) \equiv [l_i; B_i\{X^+\}_{i \in 1..n}]$, and set, with some overloading of notation:

$$a.l_j \triangleq$$

(use a as $Z <: A$, $y:A(Z)$ in $y.l_j : B_i\{A^+\}$)

$$a.l_j \leqslant \zeta(Y <: A, y:A(Y), x:A(Y))b\{Y, y, x\} \triangleq$$

(use a as $Z <: A$, $y:A(Z)$ in $\zeta(Y <: A = Z)(y.l_j \leqslant \zeta(x:A(Y))b\{Y, y, x\}) : A$)

The ζ -object selection operation reduces fairly simply to a regular selection operation on the underlying object.

The ζ -object override operation is more interesting, although similarly it reduces to an override operation on the underlying object. The overriding method b can take advantage of three variables: (1) $Y <: A$, the unknown subtype of A that was used to construct a ; (2) $y:A(Y)$,

the raw object inside a , which can be thought of as the *old self*; y is the value of self at the time the overriding takes place, containing the old version of method l_j ; (3) $x:A(Y)$, the regular self of the overriding method b . From these variables, b must produce a result of type $B_j\{Y^+\}$, parametrically in Y .

Combining the rules for objects and for Self quantification with the definitions above, we derive the rules given in appendix B.

The most remarkable fact is that the (Sub ζ Object) rule holds for ζ -object types. We recall that in section 2.5 we found that the analogous rule for recursive object types did not hold.

The (Val ζ Object) rule can be used to build a ζ -object $\zeta(Y <: A = C)b\{Y\}$ from a subtype C of the desired ζ -object type A , and from a regular object $b\{C\}$. The Y variable in $b\{Y\}$ is the Self type, in case the methods of b need to refer to it.

When building a ζ -object by (Val ζ Object) we can take $C <: A$ to be a concrete object type, often choosing $C = A$; we rarely need to work parametrically for an arbitrary $X <: A$. However, the flexibility of using an arbitrary subtype of A is critical in the derivation of (Val ζ Override). In section 5.1 we will see that this flexibility has a price.

The (Eq Sub ζ Object) rule is of limited power because the same C appears on both sides of the conclusion. We can trace back this limitation to a similar one in the rules for existentials.

4.2 Examples

We are now ready to examine some object-oriented examples (cf. [Abadi, Cardelli 1994c]). We find that these examples can be typed rather easily when seen in terms of ζ -objects, even when a method needs to return or to modify self. When constructing a fixed ζ -object, its methods are not required to operate on an arbitrary self: they just need to match the given representation type of the object being constructed. That is, to construct a ζ -object of type $A \equiv \zeta(X^+)[l_i:B_i\{X\} \ i \in 1..n]$ we need only a set of methods $b_j:B_j\{A^+\}$ (not $b_j:B_j\{X^+\}$ for an arbitrary $X <: A$). Moreover, each of these methods can assume the existence a self parameter $x_j:l_j:B_j\{A^+\} \ i \in 1..n$. (See the rules (Val ζ Object) and (Val Object).)

4.2.1 Movable Points

This is a modified version of the problematic example of section 2.5, obtained by replacing μ with ζ . We define the types of one-dimensional and two-dimensional movable points:

$$\begin{aligned} P_1 &\triangleq \zeta(\text{Self}^+)[x:\text{Int}, \text{mv_x}:\text{Int} \rightarrow \text{Self}] \\ P_2 &\triangleq \zeta(\text{Self}^+)[x,y:\text{Int}, \text{mv_x}, \text{mv_y}:\text{Int} \rightarrow \text{Self}] \end{aligned}$$

We have the desirable property $P_2 <: P_1$, by (Sub ζ Object).

Next we define the one-dimensional origin point, where Self is P_1 :

$$\text{origin}_1 : P_1 \triangleq \zeta(\text{Self} = P_1) [x=0, \text{mv_x} = \zeta(s:P_1(\text{Self}))\lambda(dx:\text{Int})\zeta(\text{Self}, s.x := s.x + dx)]$$

The rule (Val ζ Select) allows us to invoke methods whose type involves Self:

$$\text{origin}_1.\text{mv_x} : \text{Int} \rightarrow P_1$$

Moreover, the equational theory allows us to derive expected equivalencies, such as:

$$\begin{aligned} \text{origin}_1.\text{mv_x}(1) \\ \leftrightarrow \zeta(\text{Self} = P_1) [x=1, \text{mv_x} = \zeta(s:P_1(\text{Self}))\lambda(dx:\text{Int})\zeta(\text{Self}, s.x := s.x + dx)] : P_1 \end{aligned}$$

that is, the unit point equals the result of moving the origin point.

4.2.2 Object-Oriented Natural Numbers

The type of Scott numerals [Wadsworth 1980] has an object-oriented counterpart:

$$N_{Ob} \triangleq \zeta(\text{Self}^+)[\text{succ:Self}, \text{case:}\forall(Z <: \text{Top})Z \rightarrow (\text{Self} \rightarrow Z) \rightarrow Z]$$

This type is well-formed because $\forall(Z <: \text{Top})Z \rightarrow (\text{X} \rightarrow Z) \rightarrow Z$ is covariant in X.

The zero numeral can be defined as:

$$\begin{aligned} \text{zero}_{Ob} : N_{Ob} &\triangleq \\ \zeta(\text{Self} = N_{Ob}) \\ [\text{case} = \lambda(Z <: \text{Top}) \lambda(z:Z) \lambda(f:\text{Self} \rightarrow Z) z, \\ \text{succ} = \zeta(n:N_{Ob}(\text{Self})) \zeta(\text{Self}, n.\text{case} := \lambda(Z <: \text{Top}) \lambda(z:Z) \lambda(f:\text{Self} \rightarrow Z) f(\zeta(\text{Self}, n))))] \end{aligned}$$

The other numerals can be obtained from zero_{Ob} . Some familiar operations are expressible:

$$\begin{aligned} \text{succ} : N_{Ob} \rightarrow N_{Ob} &\triangleq \lambda(n:N_{Ob}) n.\text{succ} \\ \text{pred} : N_{Ob} \rightarrow N_{Ob} &\triangleq \lambda(n:N_{Ob}) n.\text{case}(N_{Ob})(\text{zero}_{Ob})(\lambda(p:N_{Ob}) p) \\ \text{iszero} : N_{Ob} \rightarrow \text{Bool} &\triangleq \lambda(n:N_{Ob}) n.\text{case}(\text{Bool})(\text{true})(\lambda(p:N_{Ob}) \text{false}) \end{aligned}$$

4.2.3 A Calculator

Our final example is that of a calculator object. We exploit the ability to override methods to record the pending arithmetic operation. When an operation add or sub is entered, the equals method is overridden with code for addition or subtraction. The first two components (arg, acc) are needed for the internal operation of the calculator, while the other four (enter, add, sub, equals) provide the user interface.

$$C = \zeta(\text{Self}^+)[\text{arg,acc: Real}, \text{enter: Real} \rightarrow \text{Self}, \text{add,sub: Self}, \text{equals: Real}]$$

By subsumption, the calculator also has type:

$$\text{Calc} = \zeta(\text{Self}^+)[\text{enter: Real} \rightarrow \text{Self}, \text{add,sub: Self}, \text{equals: Real}]$$

This shorter Calc type is the one shown to users of the calculator.

$$\begin{aligned} \text{calculator: C} &\triangleq \\ \zeta(\text{Self} = C) \\ [\text{arg} = 0.0, \\ \text{acc} = 0.0, \\ \text{enter} = \zeta(s:C(\text{Self})) \lambda(n:\text{Real}) \zeta(\text{Self}, s.\text{arg} := n), \\ \text{add} = \zeta(s:C(\text{Self})) \zeta(\text{Self}, (s.\text{acc} := s.\text{equals}).\text{equals} \Leftarrow \zeta(s':C(\text{Self})) s'.\text{acc} + s'.\text{arg}), \\ \text{sub} = \zeta(s:C(\text{Self})) \zeta(\text{Self}, (s.\text{acc} := s.\text{equals}).\text{equals} \Leftarrow \zeta(s':C(\text{Self})) s'.\text{acc} - s'.\text{arg}), \\ \text{equals} = \zeta(s:C(\text{Self})) s.\text{arg}] \end{aligned}$$

This definition is meant to provide the following behavior:

$$\begin{aligned} \text{calculator .enter}(5.0) .\text{equals} &= 5.0 \\ \text{calculator .enter}(5.0) .\text{sub .enter}(3.5) .\text{equals} &= 1.5 \\ \text{calculator .enter}(5.0) .\text{add .add .equals} &= 15.0 \end{aligned}$$

A scientific calculator can also be defined, with additional state and operations. Its inner design could be quite different from that of our basic calculator, but the scientific calculator's type may still be a subtype of Calc.

5. Overriding and Self

In this section we discuss attempts to override methods that return self.

If we want to override a method of a ζ -object of type A, the new method must work for any possible subtype of A. This is because the ζ -object might have been constructed as an element of an unknown proper subtype of A. If the new method returns self, it is critical that the type of its result be the unknown subtype of A, because one of the other methods may be invoked on the result. We say that the overriding method must be “parametric in self”; this turns out to be a difficult criterion to meet.

It should not be too surprising that it is hard to override methods that return self. After all, the technique for obtaining ζ -object subtypings is based on that of section 3.3 for obtaining covariant object components, which cannot be usefully overridden.

5.1 Overriding from the Outside

At the beginning of section 4.2 we remark how easy it is to create a ζ -object, because its initial methods needs to work only for the actual type of the object being constructed. In particular, methods that override self present no difficulties. However, if we want to override a method of an existing ζ -object $o:A$, the new method must work for any possible $B <: A$, because o might have been built as an element of B . We do not know either the “true type” of o , or the “true type” of the self parameters of its methods. When overriding a method of o , the overriding method can assume only that the object has been constructed from an unknown $\text{Self} <: A$. The same difficulty would likely surface at object-creation time, if we were creating objects incrementally, adding methods to an empty object, instead of creating full objects at once.

This is where we need the complex derived rule for overriding ζ -objects, (Val ζ Override). Consider, for example, the type:

$$Q \triangleq \zeta(\text{Self}^+)[n, f: \text{Int}, m: \text{Self}] \quad \text{with} \quad Q(X) \equiv [n, f: \text{Int}, m: X]$$

An overriding method for $o = \zeta(Y <: Q = C)b\{Y\}$ can use in its body the variables $\text{Self} <: Q$, $x' : Q(\text{Self})$, and $x : Q(\text{Self})$, where x' is in fact $b\{C\}$, according to (Eval ζ Override), and x is the self of the new method. We can therefore override the f method of Q with any of the following method bodies:

$$\begin{aligned} o.f &\Leftarrow \zeta(\text{Self} <: Q, x' : Q(\text{Self}), x : Q(\text{Self})) \quad (\text{any of the following:}) \\ &x'.n + 1 \quad \text{setting } o.f \text{ to produce } b.n + 1 \text{ (constantly)} \\ &x.n + 1 \quad \text{setting } o.f \text{ to produce } 1 + \text{the value of } n \text{ when } f \text{ is invoked} \\ &(x.n := x'.n).n \quad \text{setting } o.f \text{ to update } n \text{ with } b.n \text{ (constantly) when } f \text{ is invoked} \end{aligned}$$

Let us now attempt to override the m method. The typing rule (Val ζ Override) requires that, from the variables $\text{Self} <: Q$, $x' : Q(\text{Self})$, $x : Q(\text{Self})$ at its disposal, the overriding method must produce a value of type Self . Here are some possibilities:

$$\begin{aligned} o.m &\Leftarrow \zeta(\text{Self} <: Q, x' : Q(\text{Self}), x : Q(\text{Self})) \quad (\text{any of the following:}) \\ &x'.m \quad \text{setting } o.m \text{ to produce the current } b.m \text{ (constantly)} \\ &x.m \quad \text{setting } o.m \text{ to diverge} \end{aligned}$$

However, we cannot override $o.m$ with anything useful. Note, first, that we cannot synthesize a value of type Self from scratch. Second, we cannot return x' or x , nor $\zeta(Q, x')$ or $\zeta(Q, x)$, because none of these can be given type Self . Third, $\zeta(\text{Self}, x) : \text{Self}$ is not derivable, for an unknown $\text{Self} <: Q$. Finally, any update to x' or x will preserve their original type, so the updated x' or x cannot be returned either.

Moreover, it would be unsound to ignore these typing problems and return, say, $\zeta(\text{Self}, x)$ or $\zeta(\text{Self}, x')$. The reason for unsoundness can be traced back to the rule for constructing ζ -objects. Because of the flexibility we have in constructing ζ -objects out of proper subtypes of their types, the x and x' parameters at our disposal when overriding may be “shorter” than the subtype used originally when constructing the object.

In conclusion, we discover that the (Val ζ Override) rule, although very powerful for overriding simple methods and fields, is not sufficient to allow us to override methods that return a value of type Self. One solution to this problem is discussed in the next section.

5.2 Recoup

In this section we introduce a special method called *recoup* with an associated run-time invariant. Recoup is a method that returns self immediately. The invariant asserts that the result of *recoup* is its host object.

5.2.1 The Recoup Method

Let us redefine the type Q of section 5.1, by adding a method called *recoup*:

$$Q \triangleq \zeta(\text{Self}^+)[\text{recoup}: \text{Self}, n, f: \text{Int}, m: \text{Self}]$$

We can build an element of Q as follows:

$$o : Q \triangleq \zeta(\text{Self} = Q) b, \quad \text{where } b \equiv [\text{recoup} = \zeta(s: Q(\text{Self})) \zeta(\text{Self}, s), n = \dots, f = \dots, m = \dots]$$

Then, we can typecheck:

$$o.m \in \zeta(\text{Self} \leq Q, s' : Q(\text{Self}), s : Q(\text{Self})) s'.\text{recoup} : Q$$

since $s'.\text{recoup}$ has type Self. Moreover, the behavior obtained could be useful, and corresponds to storing the current object into the new object (like a “backup” operation).

We say that a method of the form $\zeta(s: B(\text{Self})) \zeta(\text{Self}, s)$, in the context of a ζ -object of the form $\zeta(\text{Self} \leq B = B) \dots$, is a *recoup* method. Intuitively, *recoup* allows us to recover a “parametric self” $s'.\text{recoup}$, which equals o but has type $\text{Self} \leq Q$ and not just type Q. This technique is particularly useful after an override on a value of type $\text{Self} \leq Q$, because the result of the override only has type Q.

In general, if B has the form $\zeta(\text{Self}^+)[\text{recoup}: \text{Self}, \dots]$ then we can write useful polymorphic functions of type $\forall(\text{Self} \leq B) B(\text{Self}) \rightarrow \text{Self}$ that are not available without *recoup*, such as:

$$\begin{aligned} g : \forall(\text{Self} \leq Q) Q(\text{Self}) \rightarrow \text{Self} \triangleq \\ \lambda(\text{Self} \leq Q) \lambda(s : Q(\text{Self})) (s.m := s.\text{recoup}).\text{recoup} \end{aligned}$$

Such functions are sufficiently parametric to be used in overrides from the outside, as in:

$$o.m \in \zeta(\text{Self} \leq Q, s' : Q(\text{Self}), s : Q(\text{Self})) g(Q)(s)$$

The technique just described gives the correct result only as long as *recoup* is bound to $\zeta(s : Q(\text{Self})) \zeta(\text{Self}, s)$. Otherwise, the operational behavior is not the expected one. The correctness of typing, on the other hand, does not depend on the *recoup* invariant.

An invariant of this kind is, we believe, perfectly acceptable for a programming language: *recoup* would be a distinguished component that is appropriately initialized and that cannot be overridden. Even without language support, we may be disciplined enough to preserve the *recoup* invariant, and thus we may solve the problem of overriding methods with result type Self.

5.2.2 Recoup as the Only True Method

Despite its trivial code, recoup has remarkable power: using recoup we can replace other proper methods with simple fields. We show this for an object type A without Self, but our technique might extend to ζ -object types. Suppose we have:

$$\begin{aligned} A &\triangleq [m:B] \\ o : A &\triangleq [m=\zeta(s:A)b] \end{aligned}$$

We rewrite A and o, introducing a Self quantifier and a recoup method, and changing the method m into a field m' containing a function. This modification requires using the general Self quantifier without covariance restrictions (from section 3.4). We take:

$$\begin{aligned} A' &\triangleq \zeta(\text{Self})[\text{recoup:Self}, m':\text{Self}\rightarrow B] \\ o' : A' &\triangleq \zeta(\text{Self}=A') [\text{recoup}=\zeta(s:A'(\text{Self}))\zeta(\text{Self}, s), m'=\lambda(s:\text{Self})b'] \end{aligned}$$

Thus the proper method $m=\zeta(s:A)b$ in o is replaced with the field $m'=\lambda(s:A'(\text{Self}))b'$ in o' . Next we consider how to simulate operations on elements of A with operations on elements of A', and specifically how to encode invocation and overriding. Using our encoding of invocation and overriding, we can find a b' that imitates b.

We cannot give a type to $o'.m'$. (Self must not escape the scope of ζ , and the contravariant occurrence of Self, unlike the covariant ones, cannot be replaced by its bound.) However, we can extract $o'.m'$ inside the scope of ζ and immediately apply it to $o'.\text{recoup}$, thus eliminating the single contravariant occurrence of Self before exiting the scope of the quantifier:

$$\text{invoke}(o', m') \triangleq \text{use } o' \text{ as } Y < A', y:A'(Y) \text{ in } y.m'(y.\text{recoup}) : B$$

If $o'.\text{recoup}$ is $\zeta(s:A'(A'))\zeta(A', s)$ as expected then $\text{invoke}(o', m')$ has the same behavior as $o.m$.

Furthermore, m' can be overridden with any function f: $\forall(Y < A') Y \rightarrow B$:

$$\text{override}(o', m', f) \triangleq \text{use } o' \text{ as } Y < A', y:A'(Y) \text{ in } \zeta(X=Y) y.m' := f(X) : A'$$

More general override constructs are available as well, because what used to be methods can now be extracted as functions within the scope of an override. For example, when we have a movable point o' of type $\zeta(\text{Self})[\text{recoup:Self}, mv_x': \text{Self}\rightarrow\text{Int}\rightarrow\text{Self}, \dots]$ we can override the move method (a function field) with one that runs the old method with a different parameter:

$$\begin{aligned} &\text{use } o' \text{ as } Y < A', y:[\text{recoup:Y}, mv_x': Y \rightarrow \text{Int}\rightarrow Y, \dots] \\ &\text{in } \zeta(\text{Self}=Y) y.mv_x' := \lambda(s:\text{Self})\lambda(n:\text{Int})y.mv_x'(s)(n+1) \\ &\quad : \zeta(\text{Self})[\text{recoup:Self}, mv_x': \text{Self}\rightarrow\text{Int}\rightarrow\text{Self}, \dots] \end{aligned}$$

Thus we can reuse the code of old methods, and not just their results, in defining new methods. This ability is reminiscent of that provided by “super” and “method wrappers” facilities.

6. Related Work

We finish with some comparisons with the most closely related work [Bruce 1993; Mitchell, Honsell, Fisher 1993], also discussed in [Abadi, Cardelli 1994c].

Mitchell *et al.* do not support subsumption but allow object extension; Bruce formalizes two distinct subtyping relations. We have fixed-size objects, and support subsumption by using a single subtyping relation. Like Mitchell *et al.* and unlike Bruce, we do not distinguish between objects and object generators, and we allow the overriding of proper methods in objects.

Many common examples can be expressed in all these systems, with some noticeable exceptions. (1) We cannot represent inheritance directly, for example to produce a color point

from an existing point, but we can imitate it.[Abadi, Cardelli 1994b]. (2) Using quantifiers and the type Self, we are able to give uniform typings for Scott-style object-oriented numerals and similar terms. They do not include universal quantifiers.

Mitchell *et al.* and Bruce present first-order systems with primitive objects and with a built-in Self type. In contrast we have a full second-order system where Self is obtained by an encoding. The rules for Self are similar in all these systems. The rules are always complex, but ours are derivable from those for elementary objects without Self. Hence, we may claim some success in explaining Self.

Acknowledgments

John Lamping prompted us to think about encoding covariant components. Gordon Plotkin suggested we should have a system with the Self quantifier as the only second-order construct.

Appendix A: Simple-Objects Fragments

These are the typing and equality rules for simple objects.

Δ_{Ob}

(Type Object) (l_i distinct)

$E \vdash B_i \quad \forall i \in 1..n$

$E \vdash [l_i; B_i]_{i \in 1..n}$

(Val Object) (where $A \equiv [l_i; B_i]_{i \in 1..n}$)

$E, x_i:A \vdash b_i : B_i \quad \forall i \in 1..n$

$E \vdash [l_i = \zeta(x_i; A)b_i]_{i \in 1..n} : A$

(Val Select)

$E \vdash a : [l_i; B_i]_{i \in 1..n} \quad j \in 1..n$

$E \vdash a.l_j : B_j$

(Val Override) (where $A \equiv [l_i; B_i]_{i \in 1..n}$)

$E \vdash a : A \quad E, x:A \vdash b : B_j \quad j \in 1..n$

$E \vdash a.l_j \Leftarrow \zeta(x:A)b : A$

$\Delta_{<\text{Ob}}$

(Sub Object) (l_i distinct)

$E \vdash B_i \quad \forall i \in 1..n+m$

$E \vdash [l_i; B_i]_{i \in 1..n+m} <: [l_i; B_i]_{i \in 1..n}$

$\Delta_{= \text{Ob}}$

(Eq Object) (where $A \equiv [l_i; B_i]_{i \in 1..n}$)

$E, x_i:A \vdash b_i \leftrightarrow b'_i : B_i \quad \forall i \in 1..n$

$E \vdash [l_i = \zeta(x_i; A)b_i]_{i \in 1..n} \leftrightarrow [l_i = \zeta(x_i; A)b'_i]_{i \in 1..n} : A$

(Eq Select)

$E \vdash a \leftrightarrow a' : [l_i; B_i]_{i \in 1..n} \quad j \in 1..n$

$E \vdash a.l_j \leftrightarrow a'.l_j : B_j$

(Eq Override) (where $A \equiv [l_i; B_i]_{i \in 1..n}$)

$E \vdash a \leftrightarrow a' : A \quad E, x:A \vdash b \leftrightarrow b' : B_j \quad j \in 1..n$

$E \vdash a.l_j \Leftarrow \zeta(x:A)b \leftrightarrow a'.l_j \Leftarrow \zeta(x:A)b' : A$

(Eval Select)	(Eval Override) (in both: $A \equiv [l_i : B_i]_{i \in 1..n}$, $a \equiv [l_i = \zeta(x_i : A')] b_i]_{i \in 1..n+m}$)
$E \vdash a : A \quad j \in 1..n$	$E \vdash a : A \quad E, x : A \vdash b : B_j \quad j \in 1..n$
$E \vdash a.l_j \leftrightarrow b_j \{x_j \leftarrow a\} : B_j$	$E \vdash a.l_j \not\equiv \zeta(x : A)b \leftrightarrow [l_i = \zeta(x_i : A') b_i]_{i \in 1..n+m - \{j\}}, l_j = \zeta(x : A') b : A$

 $\Delta_{=Ob <}$

(Eq Sub Object) (where $A \equiv [l_i : B_i]_{i \in 1..n}$, $A' \equiv [l_i : B_i]_{i \in 1..n}, l_j : B_j]_{j \in n+1..m}$)
$E, x_i : A \vdash b_i : B_i \quad \forall i \in 1..n \quad E, x_j : A' \vdash b_j : B_j \quad \forall j \in n+1..m$
$E \vdash [l_i = \zeta(x_i : A) b_i]_{i \in 1..n} \leftrightarrow [l_i = \zeta(x_i : A') b_i]_{i \in 1..n+m} : A$

Appendix B: ζ -Objects Fragments (Derived Rules)

These are derived rules for the combination of simple objects and the Self quantifier.

 Δ_{S+}

(Type ζ Object)	(Sub ζ Object) (l_i distinct)
$E, X < \text{Top} \vdash B_i \{X^+\} \quad \forall i \in 1..n$	$E, X < \text{Top} \vdash B_i \{X^+\} \quad \forall i \in 1..n+m$
$E \vdash \zeta(X^+) [l_i : B_i \{X\}_{i \in 1..n}] \quad E \vdash \zeta(X^+) [l_i : B_i \{X\}_{i \in 1..n+m}] < \zeta(X^+) [l_i : B_i \{X\}_{i \in 1..n}]$	
(Val ζ Object) (where $A \equiv \zeta(X^+) [l_i : B_i \{X\}_{i \in 1..n}]$)	
$E \vdash C < A \quad E \vdash b \{C\} : A(C)$	
$E \vdash \zeta(Y < A = C) b \{Y\} : A$	
(Val ζ Select)	(Val ζ Override)
$E \vdash a : A \quad j \in 1..n$	$E \vdash a : A \quad E, Y < A, y : A(Y), x : A(Y) \vdash b : B_j \{Y^+\} \quad j \in 1..n$
$E \vdash a.l_j : B_j \{A^+\}$	$E \vdash a.l_j \not\equiv \zeta(Y < A, y : A(Y), x : A(Y)) b : A$

 Δ_{S+}

(Eq ζ Object) (where $A \equiv \zeta(X^+) [l_i : B_i \{X\}_{i \in 1..n}]$, $A' \equiv \zeta(X^+) [l_i : B_i \{X\}_{i \in 1..n+m}]$)
$E \vdash C < A' \quad E \vdash b \{C\} \leftrightarrow b' \{C\} : A'(C)$
$E \vdash \zeta(Y < A = C) b \{Y\} \leftrightarrow \zeta(Y < A' = C) b' \{Y\} : A$
(Eq Sub ζ Object) (where $A \equiv \zeta(X^+) [l_i : B_i \{X\}_{i \in 1..n}]$, $A' \equiv \zeta(X^+) [l_i : B_i \{X\}_{i \in 1..n+m}]$)
$E \vdash C < A' \quad E, x_i : A(C) \vdash b_i \{C\} : B_i \{C\} \quad \forall i \in 1..n \quad E, x_j : A'(C) \vdash b_j \{C\} : B_j \{C\} \quad \forall j \in n+1..m$
$E \vdash \zeta(Y < A = C) [l_i = \zeta(x_i : A(Y)) b_i \{Y\}_{i \in 1..n}] \leftrightarrow \zeta(Y < A' = C) [l_i = \zeta(x_i : A'(Y)) b_i \{Y\}_{i \in 1..n+m}] : A$
(Eq ζ Select) (where $A \equiv \zeta(X^+) [l_i : B_i \{X\}_{i \in 1..n}]$)
$E \vdash a \leftrightarrow a' : A \quad j \in 1..n$
$E \vdash a.l_j \leftrightarrow a'.l_j : B_j \{A^+\}$
(Eq ζ Override) (where $A \equiv \zeta(X^+) [l_i : B_i \{X\}_{i \in 1..n}]$)
$E \vdash a \leftrightarrow a' : A \quad E, Y < A, y : A(Y), x : A(Y) \vdash b \leftrightarrow b' : B_j \{Y^+\} \quad j \in 1..n$
$E \vdash a.l_j \not\equiv \zeta(Y < A, y : A(Y), x : A(Y)) b \leftrightarrow a'.l_j \not\equiv \zeta(Y < A, y : A(Y), x : A(Y)) b' : A$

(Eval ζ Select) (where $A \equiv \zeta(X^+)[l_i : B_i\{X\}_{i \in 1..n}]$, $c \equiv \zeta(Z <: A = C)a\{Z\}$)
$E \vdash c : A \quad j \in 1..n$
$E \vdash c.l_j \leftrightarrow a\{C\}.l_j : B_j\{A^+\}$
(Eval ζ Override) (where $A \equiv \zeta(X^+)[l_i : B_i\{X\}_{i \in 1..n}]$, $c \equiv \zeta(Z <: A = C)a\{Z\}$)
$E \vdash c : A \quad E, Y <: A, y:A(Y), x:A(Y) \vdash b\{Y, y, x\} : B_j\{Y^+\} \quad j \in 1..n$
$E \vdash c.l_j \not\equiv \zeta(Y <: A, y:A(Y), x:A(Y))b\{Y, y, x\} \leftrightarrow \zeta(Z <: A = C)a\{Z\}.l_j \not\equiv \zeta(x:A(Z))b\{Z, a\{Z\}, x\} : A$

Appendix C: Other Typing Fragments

Δ_x

$(Env \emptyset)$	$(Env x)$	$(Val x)$
$\frac{}{\emptyset \vdash \diamond}$	$\frac{(Env x) \quad E \vdash A \quad x \notin \text{dom}(E)}{E, x:A \vdash \diamond}$	$\frac{}{E', x:A, E'' \vdash \diamond}$
		$E', x:A, E'' \vdash x:A$

Δ_K

$(Type\ Const)$	$(Val\ Const)$
$E \vdash \diamond \quad K \in \text{Sort}$	$k \in \text{Op}(K_i _{i \in 1..n+1}) \quad E \vdash a_i : K_i \quad \forall i \in 1..n$
$E \vdash K \quad E \vdash k(a_i _{i \in 1..n}) : K_{n+1}$	

Δ_{\rightarrow}

$(Type\ Arrow)$	$(Val\ Fun)$	$(Val\ Appl)$
$E \vdash A \quad E \vdash B$	$E, x:A \vdash b : B$	$E \vdash b : A \rightarrow B \quad E \vdash a : A$
$E \vdash A \rightarrow B$	$E \vdash \lambda(x:A)b : A \rightarrow B$	$E \vdash b(a) : B$

$\Delta_{<}$

$(Sub\ Refl)$	$(Sub\ Trans)$	$(Val\ Subsumption)$
$E \vdash A$	$E \vdash A < B \quad E \vdash B < C$	$E \vdash a : A \quad E \vdash A < B$
$E \vdash A <: A \quad E \vdash A <: C \quad E \vdash a : B$		
$(Type\ Top)$	$(Sub\ Top)$	
$E \vdash \diamond$	$E \vdash A$	
$E \vdash \text{Top}$	$E \vdash A < \text{Top}$	

$\Delta_{<:K}$

$(Sub\ Sort)$
$E \vdash \diamond \quad (K, K') \in \text{SubSort}$
$E \vdash K < K'$

$\Delta_{\leq\rightarrow}$

(Sub Arrow)

$$\frac{E \vdash A' <: A \quad E \vdash B <: B'}{E \vdash A \rightarrow B <: A' \rightarrow B'}$$

 $\Delta_{\leq X}$ (Env $X <:$)

$$\frac{E \vdash A \quad X \notin \text{dom}(E)}{E, X <: A \vdash \diamond}$$

(Type $X <:$)

$$\frac{E', X <: A, E'' \vdash \diamond}{E', X <: A, E'' \vdash X}$$

(Sub X)

$$\frac{}{E', X <: A, E'' \vdash X <: A}$$

 $\Delta_{\leq \mu}$ (Type Rec $<:$)

$$\frac{E, X <: \text{Top} \vdash A \quad A > X}{E \vdash \mu(X)A}$$

(Sub Rec)

$$\frac{E \vdash \mu(X)A \quad E \vdash \mu(Y)B \quad E, Y <: \text{Top}, X <: Y \vdash A <: B}{E \vdash \mu(X)A <: \mu(Y)B}$$

(Val Fold)

$$\frac{E \vdash a : A \{X \leftarrow \mu(X)A\}}{E \vdash \text{fold}(\mu(X)A, a) : \mu(X)A}$$

(Val Unfold)

$$\frac{E \vdash a : \mu(X)A}{E \vdash \text{unfold}(a) : A \{X \leftarrow \mu(X)A\}}$$

(Val Rec)

$$\frac{E, x : A \vdash a : A}{E \vdash \mu(x : A)a : A}$$

 $\Delta_{\leq \forall}$ (Type All $<:$)

$$\frac{E, X <: A \vdash B}{E \vdash \forall(X <: A)B}$$

(Sub All)

$$\frac{E \vdash A' <: A \quad E, X <: A' \vdash B <: B'}{E \vdash \forall(X <: A')B <: \forall(X <: A')B'}$$

(Val Fun2 $<:$)

$$\frac{E, X <: A \vdash b : B}{E \vdash \lambda(X <: A)b : \forall(X <: A)B}$$

(Val App2 $<:$)

$$\frac{E \vdash b : \forall(X <: A)B \{X\} \quad E \vdash A' <: A}{E \vdash b(A') : B \{A'\}}$$

 $\Delta_{\leq \exists}$ (Type Exists $<:$)

$$\frac{E, X <: A \vdash B}{E \vdash \exists(X <: A)B}$$

(Sub Exists)

$$\frac{E \vdash A <: A' \quad E, X <: A \vdash B <: B'}{E \vdash \exists(X <: A)B <: \exists(X <: A')B'}$$

(Val Pack $<:$)

$$\frac{E \vdash C <: A \quad E \vdash b\{C\} : B\{C\}}{E \vdash (\text{pack } X <: A = C, b\{X\} : B\{X\}) : \exists(X <: A)B\{X\}}$$

(Val Open $<:$)

$$\frac{E \vdash c : \exists(X <: A)B \quad E \vdash D \quad E, X <: A, x : B \vdash d : D}{E \vdash (\text{open } c \text{ as } X <: A, x : B \text{ in } d : D) : D}$$

The relation $A > Y$ (type expression A is formally contractive in variable Y) is:

$X > Y$	if $X \neq Y$
$\text{Top} > Y$	always
$\llbracket \cdot ; B_i \mid_{i \in 1..n} \rrbracket > Y$	always
$A \rightarrow B > Y$	always
$\mu(X)A > Y$	if $A > Y$
$\forall(X <: A)B > Y$	if $B > X$ and $B > Y$ (no requirement on A)
$\exists(X <: A)B > Y$	if $B > X$ and $B > Y$ (no requirement on A)

Appendix D: Other Equational Fragments

Δ_{\sim}

(Eq Symm)	(Eq Trans)
$E \vdash a \leftrightarrow b : A$	$E \vdash a \leftrightarrow b : A \quad E \vdash b \leftrightarrow c : A$
$E \vdash b \leftrightarrow a : A$	$E \vdash a \leftrightarrow c : A$

$\Delta_{=x}$

(Eq x)
$E', x:A, E'' \vdash \diamond$
$E', x:A, E'' \vdash x \leftrightarrow x : A$

$\Delta_{=K}$

(Eq Const)
$k \in \text{Op}(K_i \mid_{i \in 1..n+1})$
$E \vdash a_i \leftrightarrow a'_i : K_i \quad \forall i \in 1..n$

$$E \vdash k(a_i \mid_{i \in 1..n}) \leftrightarrow k(a'_i \mid_{i \in 1..n}) : K_{n+1}$$

$\Delta_{\sim \rightarrow}$

(Eq Fun)	(Eq Appl)
$E, x:A \vdash b \leftrightarrow b' : B$	$E \vdash b \leftrightarrow b' : A \rightarrow B \quad E \vdash a \leftrightarrow a' : A$
$E \vdash \lambda(x:A)b \leftrightarrow \lambda(x:A)b' : A \rightarrow B$	$E \vdash b(a) \leftrightarrow b'(a') : B$
(Eval Beta)	(Eval Eta)
$E \vdash \lambda(x:A)b : A \rightarrow B \quad E \vdash a : A$	$E \vdash b : A \rightarrow B \quad x \notin \text{dom}(E)$
$E \vdash (\lambda(x:A)b)(a) \leftrightarrow b\{x \leftarrow a\} : B$	$E \vdash \lambda(x:A)b(x) \leftrightarrow b : A \rightarrow B$

$\Delta_{\sim \leftarrow}$

(Eq Subsumption)	(Eq Top)
$E \vdash a \leftrightarrow a' : A \quad E \vdash A <: B$	$E \vdash a:A \quad E \vdash b:B$
$E \vdash a \leftrightarrow a' : B$	$E \vdash a \leftrightarrow b : \text{Top}$

$\Delta_{\equiv\mu<}$

(Eq Fold)	(Eq Unfold)
$E \vdash a \leftrightarrow a' : A\{X \leftarrow \mu(X)A\}$	$E \vdash a \leftrightarrow a' : \mu(X)A$
$E \vdash \text{fold}(\mu(X)A, a) \leftrightarrow \text{fold}(\mu(X)A, a') : \mu(X)A \quad E \vdash \text{unfold}(a) \leftrightarrow \text{unfold}(a') : A\{X \leftarrow \mu(X)A\}$	
(Eq Fold<:)	
$E \vdash \mu(X)A \quad E \vdash \mu(Y)B \quad E, Y <: \text{Top}, X <: Y \vdash A <: B \quad E \vdash a \leftrightarrow a' : A\{X \leftarrow \mu(X)A\}$	$E \vdash \text{fold}(\mu(X)A, a) \leftrightarrow \text{fold}(\mu(Y)B, a') : \mu(Y)B$
(Eval Fold)	(Eval Unfold)
$E \vdash a : \mu(X)A$	$E \vdash a : A\{X \leftarrow \mu(X)A\}$
$E \vdash \text{fold}(\mu(X)A, \text{unfold}(a)) \leftrightarrow a : \mu(X)A \quad E \vdash \text{unfold}(\text{fold}(\mu(X)A, a)) \leftrightarrow a : A\{X \leftarrow \mu(X)A\}$	
(Eq Rec)	(Eval Rec)
$E, x:A \vdash a \leftrightarrow a' : A$	$E, x:A \vdash a : A$
$E \vdash \mu(x:A)a \leftrightarrow \mu(x:A)a' : A$	$E \vdash \mu(x:A)a \leftrightarrow a\{x \leftarrow \mu(x:A)a\} : A$

 $\Delta_{\equiv\forall}$

(Eq Fun2<:)	(Eq Appl2<:)
$E, X <: A \vdash b \leftrightarrow b' : B$	$E \vdash b \leftrightarrow b' : \forall(X <: A)B\{X\} \quad E \vdash A' <: A$
$E \vdash \lambda(X <: A)b \leftrightarrow \lambda(X <: A)b' : \forall(X <: A)B$	
(Eval Beta2<:)	(Eval Eta2<:)
$E \vdash \lambda(X <: A)b : \forall(X <: A)B \quad E \vdash C <: A$	$E \vdash b : \forall(X <: A)B \quad X \notin \text{dom}(E)$
$E \vdash (\lambda(X <: A)b)(C) \leftrightarrow b\{X \leftarrow C\} : B\{X \leftarrow C\}$	$E \vdash \lambda(X <: A)b(X) \leftrightarrow b : \forall(X <: A)B$

 $\Delta_{\equiv\exists}$

(Eq Pack<:)	
$E \vdash C <: A' \quad E \vdash A' <: A \quad E, X <: A' \vdash B'\{X\} <: B\{X\} \quad E \vdash b\{C\} \leftrightarrow b'\{C\} : B'\{C\}$	
$E \vdash (\text{pack } X <: A = C, b\{X\} : B\{X\}) \leftrightarrow (\text{pack } X <: A' = C, b'\{X\} : B'\{X\}) : \exists(X <: A)B\{X\}$	
(Eq Open<:)	
$E \vdash c \leftrightarrow c' : \exists(X <: A)B \quad E \vdash D \quad E, X <: A, x:B \vdash d \leftrightarrow d' : D$	
$E \vdash (\text{open } c \text{ as } X <: A, x:B \text{ in } d:D) \leftrightarrow (\text{open } c' \text{ as } X <: A, x:B \text{ in } d':D) : D$	
(Eval Unpack<:) (where $c \equiv \text{pack } X <: A = C, b\{X\} : B\{X\}$)	
$E \vdash c : \exists(X <: A)B\{X\} \quad E \vdash D \quad E, X <: A, x:B\{X\} \vdash d\{X, x\} : D$	
$E \vdash (\text{open } c \text{ as } X <: A, x:B\{X\} \text{ in } d\{X, x\}:D) \leftrightarrow d\{C, b\{C\}\} : D$	
(Eval Repack<:)	
$E \vdash b : \exists(X <: A)B\{X\} \quad E, y:\exists(X <: A)B\{X\} \vdash d\{y\} : D$	
$E \vdash (\text{open } b \text{ as } X <: A, x:B\{X\} \text{ in } d\{\text{pack } X' <: A = X, x:B\{X'\}\}:D) \leftrightarrow d\{b\} : D$	

Appendix E: The ζ Ob Calculus

ζ Ob is our minimal second-order object calculus. It is obtained by combining the rules for object types (appendix A) with the Self quantifier (section 3.4) taken as a primitive, plus some general rules (appendix C and D). The rules for ζ -objects (appendix B) are derivable from the ones shown here. The relation $A > X$ is defined as in appendix C, with in addition $\zeta(X)B > Y$ if $B > Y$.

$$A,B ::= X \mid \text{Top} \mid [l_i:B_i]_{i \in 1..n} \mid \zeta(X)B$$

$$a,b ::= x \mid [l_i=\zeta(x_i:A)b_i]_{i \in 1..n} \mid a.l \mid a.l \leq \zeta(x:A)b \mid \zeta(X <: A=B)b \mid \text{use } a \text{ as } X <: A, y:B \text{ in } b:D$$

(Env \emptyset)	(Env x)	(Env $X <:$)
$\frac{}{\emptyset \vdash \diamond}$	$\frac{(E \vdash A \quad x \notin \text{dom}(E))}{E,x:A \vdash \diamond}$	$\frac{(E \vdash A \quad X \notin \text{dom}(E))}{E,X <: A \vdash \diamond}$

(Type $X <:$)	(Type Top)	(Type Object) (l_i distinct)	(Type Self)
$E',X <: A,E'' \vdash \diamond$	$E \vdash \diamond$	$E \vdash B_i \quad \forall i \in 1..n$	$E,X <: \text{Top} \vdash B \quad B > X$
$E',X <: A,E'' \vdash X$	$E \vdash \text{Top}$	$E \vdash [l_i:B_i]_{i \in 1..n}$	$E \vdash \zeta(X)B$

(Sub Refl)	(Sub Trans)	(Sub X)
$E \vdash A$	$\frac{E \vdash A <: B \quad E \vdash B <: C}{E \vdash A <: C}$	$E',X <: A,E'' \vdash \diamond$
$E \vdash A <: A$	$E \vdash A <: C$	$E',X <: A,E'' \vdash X <: A$
(Sub Top)	(Sub Object) (l_i distinct)	(Sub Self)
$E \vdash A$	$E \vdash B_i \quad \forall i \in 1..n+m$	$E,X <: \text{Top} \vdash B <: B' \quad B,B' > X$
$E \vdash A <: \text{Top}$	$E \vdash [l_i:B_i]_{i \in 1..n+m} <: [l_i:B_i]_{i \in 1..n}$	$E \vdash \zeta(X)B <: \zeta(X)B'$

(Val Subsumption)	(Val x)	(Val Object) (where $A \equiv [l_i:B_i]_{i \in 1..n}$)
$E \vdash a : A \quad E \vdash A <: B$	$E',x:A,E'' \vdash \diamond$	$E, x_i:A \vdash b_i : B_i \quad \forall i \in 1..n$
$E \vdash a : B$	$E',x:A,E'' \vdash x:A$	$E \vdash [l_i=\zeta(x_i:A)b_i]_{i \in 1..n} : A$
(Val Select)	(Val Override) (where $A \equiv [l_i:B_i]_{i \in 1..n}$)	
$E \vdash a : [l_j:B_j]_{j \in 1..n}$	$E \vdash a : A \quad E, x:A \vdash b : B_j \quad j \in 1..n$	
$E \vdash a.l_j : B_j$	$E \vdash a.l_j \leq \zeta(x:A)b : A$	
(Val Self) (where $A \equiv \zeta(X)B\{X\}$)	(Val Use) (where $A \equiv \zeta(X)B\{X\}$)	
$E \vdash C <: A \quad E \vdash b(C) : B\{C\}$	$E \vdash c : A \quad E \vdash D \quad E, Y <: A, y:B\{Y\} \vdash d : D$	
$E \vdash \zeta(Y <: A=C) b(Y) : A$	$E \vdash (\text{use } c \text{ as } Y <: A, y:B\{Y\} \text{ in } d:D) : D$	

(Eq Symm)	(Eq Trans)
$E \vdash a \leftrightarrow b : A$	$\frac{E \vdash a \leftrightarrow b : A \quad E \vdash b \leftrightarrow c : A}{E \vdash a \leftrightarrow c : A}$
$E \vdash b \leftrightarrow a : A$	

<p>(Eq Subsumption)</p> $\frac{E \vdash a \leftrightarrow a' : A \quad E \vdash A <: B}{E \vdash a \leftrightarrow a' : B}$	<p>(Eq x)</p> $\frac{E', x:A, E'' \vdash \diamond}{E', x:A, E'' \vdash x \leftrightarrow x : A}$	<p>(Eq Top)</p> $\frac{E \vdash a:A \quad E \vdash b:B}{E \vdash a \leftrightarrow b : \text{Top}}$
(Eq Object) (where $A \equiv [l_i:B_i]_{i \in 1..n}]$)		
$\frac{E, x_i:A \vdash b_i \leftrightarrow b'_i : B_i \quad \forall i \in 1..n}{E \vdash [l_i = \zeta(x_i:A)b_i]_{i \in 1..n} \leftrightarrow [l_i = \zeta(x_i:A)b'_i]_{i \in 1..n} : A}$		
(Eq Sub Object) (where $A \equiv [l_i:B_i]_{i \in 1..n}], A' \equiv [l_i:B_i]_{i \in 1..n}, l_j:B_j]_{j \in n+1..m}]$)		
$\frac{E, x_i:A \vdash b_i : B_i \quad \forall i \in 1..n \quad E, x_j:A' \vdash b_j : B_j \quad \forall j \in n+1..m}{E \vdash [l_i = \zeta(x_i:A)b_i]_{i \in 1..n} \leftrightarrow [l_i = \zeta(x_i:A')b_i]_{i \in 1..n+m} : A}$		
<p>(Eq Select)</p> $\frac{E \vdash a \leftrightarrow a' : [l_i:B_i]_{i \in 1..n} \quad j \in 1..n}{E \vdash a.l_j \leftrightarrow a'.l_j : B_j}$	<p>(Eq Override) (where $A \equiv [l_i:B_i]_{i \in 1..n}]$)</p> $\frac{E \vdash a \leftrightarrow a' : A \quad E, x:A \vdash b \leftrightarrow b' : B_j \quad j \in 1..n}{E \vdash a.l_j \not\equiv \zeta(x:A)b \leftrightarrow a'.l_j \not\equiv \zeta(x:A)b' : A}$	
(Eq Self) (where $A \equiv \zeta(X)B\{X\}, A' \equiv \zeta(X)B'\{X\}$)		
$\frac{E \vdash C <: A' \quad E, X <: \text{Top} \vdash B'\{X\} <: B\{X\} \quad E \vdash b\{C\} \leftrightarrow b'\{C\} : B'\{C\}}{E \vdash \zeta(Y <: A = C)b\{Y\} \leftrightarrow \zeta(Y <: A' = C)b'\{Y\} : A}$		
(Eq Use) (where $A \equiv \zeta(X)B\{X\}$)		
$\frac{E \vdash c \leftrightarrow c' : A \quad E \vdash D \quad E, Y <: A, y:B\{Y\} \vdash d \leftrightarrow d' : D}{E \vdash (\text{use } c \text{ as } Y <: A, y:B\{Y\} \text{ in } d:D) \leftrightarrow (\text{use } c' \text{ as } Y <: A, y:B\{Y\} \text{ in } d':D) : D}$		

<p>(Eval Select) (where $A \equiv [l_i:B_i]_{i \in 1..n}], a \equiv [l_i = \zeta(x_i:A')b_i]_{i \in 1..n+m}]$)</p> $\frac{E \vdash a : A \quad j \in 1..n}{E \vdash a.l_j \leftrightarrow b_j[x_j \leftarrow a] : B_j}$
<p>(Eval Override) (where $A \equiv [l_i:B_i]_{i \in 1..n}], a \equiv [l_i = \zeta(x_i:A')b_i]_{i \in 1..n+m}]$)</p> $\frac{E \vdash a : A \quad E, x:A \vdash b : B_j \quad j \in 1..n}{E \vdash a.l_j \not\equiv \zeta(x:A)b \leftrightarrow [l_i = \zeta(x_i:A')b_i]_{i \in (1..n+m)-j}, l_j = \zeta(x:A')b : A}$
<p>(Eval Unself) (where $A \equiv \zeta(X)B\{X\}, c \equiv \zeta(Z <: A = C)b\{Z\}$)</p> $\frac{E \vdash c : A \quad E \vdash D \quad E, Y <: A, y:B\{Y\} \vdash d\{Y, y\} : D}{E \vdash (\text{use } c \text{ as } Y <: A, y:B\{Y\} \text{ in } d\{Y, y\}:D) \leftrightarrow d\{C, b\{C\}\} : D}$
<p>(Eval Reself) (where $A \equiv \zeta(X)B\{X\}$)</p> $\frac{E \vdash b : A \quad E, y:A \vdash d\{y\} : D}{E \vdash (\text{use } b \text{ as } Y <: A, y:B\{Y\} \text{ in } d\{\zeta(Y <: A = Y)y\} : A) \leftrightarrow d\{b\} : D}$

References

- [Abadi, Cardelli 1994a] M. Abadi and L. Cardelli, **A semantics of object types**. *To appear*.
- [Abadi, Cardelli 1994b] M. Abadi and L. Cardelli, **A theory of primitive objects**. *To appear*.
- [Abadi, Cardelli 1994c] M. Abadi and L. Cardelli. **A theory of primitive objects: untyped and first-order systems**. *Proc. Theoretical Aspects of Computer Software*. Springer-Verlag.
- [Böhm, Berarducci 1985] C. Böhm and A. Berarducci, **Automatic synthesis of typed λ -programs on term algebras**. *Theoretical Computer Science* **39**, 135-154.
- [Bruce 1993] K. Bruce. **A paradigmatic object-oriented programming language: design, static typing, and semantics**. Technical Report No. CS-92-01, revised (to appear in the Journal of Functional Programming). Williams College.
- [Cardelli 1991] L. Cardelli. **Extensible records in a pure calculus of subtyping**. Technical Report n.81. DEC Systems Research Center.
- [Cardelli, *et al.* 1991] L. Cardelli, J.C. Mitchell, S. Martini, and A. Scedrov. **An extension of system F with subtyping**. *Proc. Theoretical Aspects of Computer Software*. Lecture Notes in Computer Science 526. Springer-Verlag.
- [Cardelli, Wegner 1985] L. Cardelli and P. Wegner, **On understanding types, data abstraction and polymorphism**. *Computing Surveys* **17**(4), 471-522.
- [Curien, Ghelli 1992] P.-L. Curien and G. Ghelli, **Coherence of subsumption, minimum typing and type-checking in F_\leq** . *Mathematical Structures in Computer Science* **2**(1), 55-91.
- [Girard, Lafont, Taylor 1989] J.-Y. Girard, Y. Lafont, and P. Taylor, **Proofs and types**. Cambridge University Press.
- [MacQueen, Plotkin, Sethi 1986] D.B. MacQueen, G.D. Plotkin, and R. Sethi, **An ideal model for recursive polymorphic types**. *Information and Control* **71**, 95-130.
- [Meyer 1988] B. Meyer, **Object-oriented software construction**. Prentice Hall.
- [Mitchell, Honsell, Fisher 1993] J.C. Mitchell, F. Honsell, and K. Fisher. **A lambda calculus of objects and method specialization**. *Proc. 8th Annual IEEE Symposium on Logic in Computer Science*.
- [Wadsworth 1980] C. Wadsworth, **Some unusual λ -calculus numeral systems**. In *To H.B. Curry: Essays on combinatory logic, lambda calculus and formalism*, J.P. Seldin and J.R. Hindley, ed. Academic Press.

Pi-nets: a graphical form of π -calculus¹

Robin Milner

Laboratory for Foundations of Computer Science
Computer Science Department, University of Edinburgh
The King's Buildings, Edinburgh EH9 3JZ, UK

Abstract An action calculus which closely corresponds to the π -calculus is presented in graphical form, as so-called *π -nets*. First an elementary form of π -net, with no sequential control, is presented. Then, using a construction by Honda and Tokoro, it is shown informally that by adding a single control construction **box** to elementary π -nets, the sequential control present in the π -calculus can be recovered. (Another construction, **rep**, provides replication.) The graphical presentation suggests a few interesting variants of this control regime, which are studied briefly. The main purpose of the paper is to explore informally the power and utility of graphical forms of the π -calculus, in the context of action calculi. It also suggests that graphical forms of other action calculi should be explored.

1 Introduction

Action structures [4] were defined as a framework for studying various notions of concurrent interactive behaviour, in the hope of yielding some taxonomy for these notions, and some uniformity in their presentation. The prime ingredients of an action structure are its *actions*. Each action a has a *source arity* m and a *target arity* n , and we write $a : m \rightarrow n$. These arities m, n are elements of a monoid, which for this paper may be taken to be the natural numbers under addition. Roughly, $a : m \rightarrow n$ is a (perhaps complex) process into which m data are fed and from which n data may be extracted. We say more later about the algebraic properties of action structures.

In a later paper [6]² an action structure called PIC was defined in the spirit of the π -calculus [8]. Roughly, PIC is what remains of the π -calculus when we remove *replication* (i.e. the power of infinite computation) and also *guarding* – represented by the dot in the process $x(y).P$ for example. An action $a : m \rightarrow n$ of PIC is essentially a collection of π -calculus particles, each being either an input particle $x(y)$, an output particle $\bar{x}(z)$ or a restriction particle νx ; it also *imports* m names and *exports* n names. PIC has a simple and illuminating graphical presentation; we call the graph which represents an action a *π -net*.

PIC is not the whole π -calculus, since it cannot simulate the power of guarding or replication; but it was later shown [7] that PIC can be freely extended within the framework of action calculi (a subclass of action structures) by the addition of two so-called *control* contructions: **box** for guarding, and **rep** for replication. (Henceforth we

¹This work was done with the support of a Senior Fellowship from the Science and Engineering Research Council, UK.

²A revised version of the two cited papers [4, 6] will appear as Parts I and II of *Action structures and the π -calculus*, in the Proceedings of the NATO Advanced Study Institute on **Proof and Computation** held at Marktoberdorf in 1993.

shall use the term “boxing” to mean some variant or other of guarding.) The resulting action calculus $\text{PIC}(\text{box}, \text{rep})$ has expressive power which is –informally speaking– comparable with that of the original π -calculus, and also has the extra structure conferred by the algebraic theory of action calculi.

The purpose of this paper is to explore and develop the *graphical* presentation of $\text{PIC}(\text{box}, \text{rep})$. We shall see that these extended π -nets throw light on the structure of interaction. In particular, the graphical presentation suggests several alternative ways in which boxing may constrain reaction.

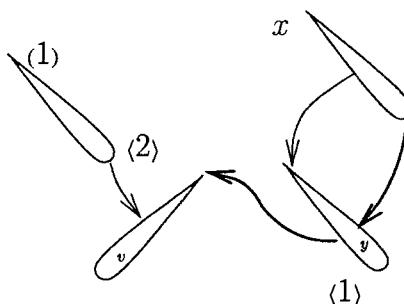
The paper can be read on its own, though the reader may find that the papers [4, 6] provide helpful background. There are similarities between π -nets and Parrow’s interaction diagrams [9], though the work was done independently. The differences mainly arise from the fact that the π -nets form an action structure.

Outline: In Section 2 we define the π -nets of PIC informally, by means of an example, and explain how they *react* (their dynamics). In Section 3 we review the notion of action structure, and define the algebraic operations upon π -nets which make PIC an action structure. In Section 4 we enrich π -nets by adding boxing and replication; this yields a version of $\text{PIC}(\text{box}, \text{rep})$ whose control regime does not permit reaction within a box. This regime is quite close to that of the original π -calculus. In Section 5 we illustrate it by performing the elegant construction by Honda and Tokoro [3], which shows that a monadic π -calculus with only input (not output) guards has the full power of polyadic π -calculus [5]. In Section 6 we relax the regime to allow reaction within a box; this is in the spirit of the *solutions* in the Chemical Abstract Machine [2], and still supports the Honda-Tokoro construction. Somewhat surprisingly, we find several variants of this relaxed regime. This is taken to be justification of the use of graphical representation; some variants are suggested by using π -nets which are not otherwise obvious. We end in Section 7 with brief reflections and suggestions for further work.

2 π -nets defined by example

The π -nets of PIC are simple graphical objects. In this section we define them informally by means of an example; we also show how reaction works in π -nets.

Below is an action $a : 1 \rightarrow 2$ in PIC , together with its formal expression:

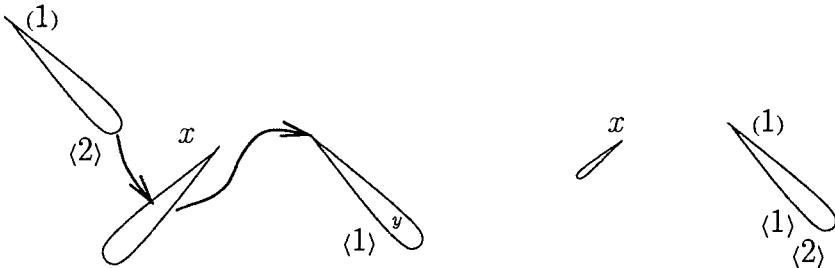


$$a = (z) [x(y) \underline{\bar{y}(x)} y(v) \bar{v}(z)] \langle yz \rangle$$

Each name (free or bound) in the expression for a corresponds to a node (a torpedo) in the π -net, so there are four nodes. But the only free name in a is x , so x is the only name which occurs properly in the π -net; it appears as a *name tag* at the *tail* (the sharp end) of its node. The labels v and y are present just to aid the eye; an arc or arrow leading to the tail of a node indicates that it corresponds to a bound name. An *imported* name such as z here is indicated by an *import tag*, e.g. $\langle 1 \rangle$; an *exported* name such as y or z here is indicated by an *export tag*, e.g. $\langle 1 \rangle$. Thus the number of such tags is determined by the source and target arities respectively.

Each arc in the π -net corresponds to an i/o particle of a . For an input particle like $y(v)$ the arc goes from the *waist* of a node to the tail of a node, while for an output particle like $\bar{y}(x)$ it goes from the *head* (the blunt end) of a node to a waist. A pair of arcs such as these which have the same *port* node (the one whose waist they impinge upon) form a *redex*; the redex is shown by underlining in the expression and by thicker lines in the net.

To *reduce* a redex, we simply remove the two arcs and coalesce the *source* and *target* nodes (x and v in this case). This forms a single node which carries all the items (arcs, tags) of both. We write $a \searrow^1 a'$ for such a single reduction, or *reaction*. The net a' which results in this case is shown in the next picture; it too has a redex, so we have another single reaction $a' \searrow^1 a''$. We show a'' too.



$$a' = (z) [\underline{x(y)} \bar{x}\langle z \rangle] \langle yz \rangle$$

$$a'' = (z) \langle zz \rangle$$

Note that the named node x in a'' is still present. Such named nodes, bearing no arcs or other tags, make no difference and can be present or absent.

We have almost fully defined π -nets, by giving the above example. A more formal definition appears in [6], and we shall not need it here; we merely emphasize the following points:

- Arcs always join a waist to a head or tail of a node.
- A node tail bears at most one of an arc, a name tag or an import tag. If it bears none of these it corresponds to a restriction particle νx in a PIC expression.
- In a π -net of arity $m \rightarrow n$, each import tag i ($1 \leq i \leq m$) or export tag j ($1 \leq j \leq n$) occurs exactly once.

The following basic axioms hold:

$$\begin{array}{ll}
 a \cdot \text{id} = a = \text{id} \cdot a & a \cdot (b \cdot c) = (a \cdot b) \cdot c \\
 a \otimes \text{id}_\epsilon = a = \text{id}_\epsilon \otimes a & a \otimes (b \otimes c) = (a \otimes b) \otimes c \\
 \text{id} \otimes \text{id} = \text{id} & (a \cdot b) \otimes (c \cdot d) = (a \otimes c) \cdot (b \otimes d) \\
 \mathbf{ab}_x \text{id} = \text{id} & \mathbf{ab}_x(a \cdot b) = (\mathbf{ab}_x a) \cdot (\mathbf{ab}_x b).
 \end{array}$$

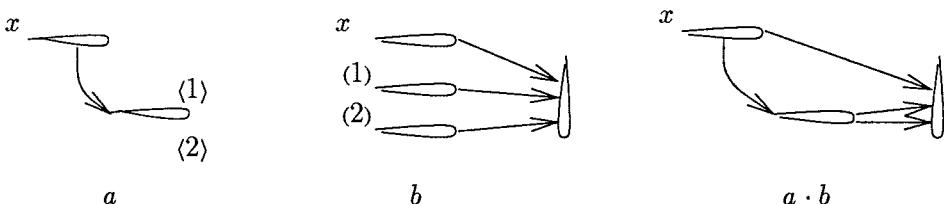
The first six axioms make A a strict monoidal category, and the last two assert that the abstractors \mathbf{ab}_x are endofunctors.

For the dynamics there is a preorder \searrow called the *reaction relation* which is preserved by product, composition and abstraction and such that if $a \searrow a'$ then a and a' have the same source and target arities, and if $\text{id} \searrow a$ then $a = \text{id}$.

We have already described the dynamics of PIC. It is straightforward to show that PIC becomes an action structure, when we define the algebraic operations as follows:

- The identity $\text{id}_m : m \rightarrow m$ consists of just m nodes, the i^{th} of which bears input tag (i) and export tag $\langle i \rangle$.
- To form $a \otimes b$, where $a : k \rightarrow \ell$, increment b 's import tags by k and export tags by ℓ , and coalesce nodes with equal name tags.
- To form $a \cdot b$, where $a : k \rightarrow \ell$ and $b : \ell \rightarrow m$, coalesce a 's node having export tag i with b 's node having import tag i (for $1 \leq i \leq \ell$), remove those tags, and coalesce nodes with equal name tags.
- To form $\mathbf{ab}_x a$, increment a 's import and export tags by 1, give the tags (1) and $\langle 1 \rangle$ to the node of a which has name tag x , and remove that name tag.

Here is an example of composition, showing both kinds of coalescence:



In fact, PIC is not only an action structure but also an *action calculus* [7], a special kind of action structure. We need not be concerned with all the details of action calculi here, but only with a key property –namely that each action calculus is generated by two special kinds of action, $\omega : 1 \rightarrow 0$ (*discard*) and $\langle x \rangle : 0 \rightarrow 1$ (*datum*, $x \in X$), which are common to all action calculi over X , and a set of actions called *controls* which are specific to each action calculus. Monadic PIC has three controls: $\text{in} : 1 \rightarrow 1$ (*input*), $\text{out} : 2 \rightarrow 0$ (*output*) and $\nu : 0 \rightarrow 1$ (*restriction*). Thus all monadic π -nets in PIC can be constructed by the action structure operations from the following generators:

Thus the waist or head of a node may bear many arcs, and the head may bear many export tags.

To appreciate the difference between π -nets and the processes of the π -calculus [8], let us compare the net $a \in \text{PIC}$ with the process P of the π -calculus in its standard notation:

$$P = x(y).(\overline{y}\langle x \rangle \mid y(v).\overline{v}\langle z \rangle).$$

There are several points:

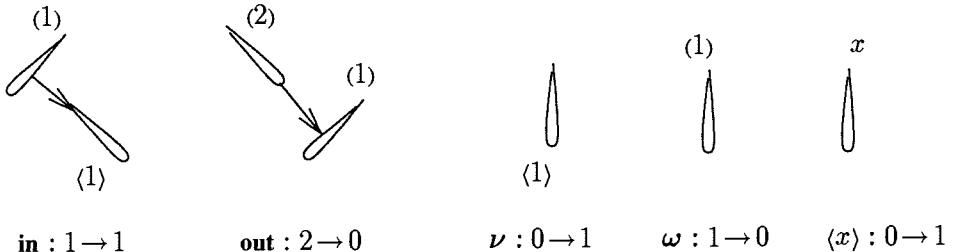
1. Note that a is abstracted (upon z), but P is not. Abstraction of port-names was not considered in the original π -calculus of [8]. Abstractions such as $\langle x \rangle P$ were introduced for the π -calculus in [5], but they were not allowed to react (as $a \in \text{PIC}$ reacts).
2. Further, a exports results (the name-vector $\langle yz \rangle$), since actions are the arrows of an action structure and so possess a target arity. A notion of *sorting* for processes was introduced in [5] –thus $\langle x \rangle P$ would have sort (1)– but this corresponds only to source arity (the number of imports), not target arity (the number of exports).
3. The input prefix $x(y)$ in P must react (with something else) before a reaction in the body of P can occur. This guarding is at once a strength and a weakness of π -calculus; it allows sequence to be imposed upon actions, but this sequential control is present for *all* i/o particles. The action structure PIC on the other hand imposes no sequencing, except that a reaction may induce an identification of names which enables another reaction. Thus the fact that the redex of a has a bound port (y) does not prevent its reaction; on the other hand the redex of a' only exists because of the first reaction.

Note that a does not include any polyadic particles, like $x(y_1 y_2)$ or $\overline{x}\langle y_1 y_2 \rangle$. Such particles were not present in the original π -calculus [8], but were introduced in [5]. To represent them in π -nets we require a kind of multi-arc. We shall have more to say later about whether multi-arcs are necessary, or whether their power can be gained by other means. For the moment we ignore them; thus we are discussing *monadic* PIC .

3 The algebra PIC and its generators

Let us briefly review the algebraic theory and dynamics of action structures [4]. Let X be a set (of names). The algebraic operations of an action structure A over X are the *identities* id_m , the binary operations *product* \otimes and *composition* \cdot , and the indexed family $\{\text{ab}_x \mid x \in X\}$ of unary operations called *abstractors*. They obey the following rules of arity, assuming that the arities are natural numbers:

$$\begin{array}{c} \text{id}_m : m \rightarrow m \\ \hline a : k \rightarrow m \qquad b : m \rightarrow p \\ \hline a \cdot b : k \rightarrow p \end{array} \qquad \begin{array}{c} a : k \rightarrow n \qquad b : \ell \rightarrow p \\ \hline a \otimes b : k \otimes \ell \rightarrow n \otimes p \end{array} \qquad \begin{array}{c} a : m \rightarrow n \\ \hline \text{ab}_x a : 1 + m \rightarrow 1 + n \end{array}.$$



In [6] it was pointed out that not *all* monadic π -nets can be so generated; no net with a *source-cycle* is generated.³ (A source cycle is a cycle in which each arc leads to a tail – i.e. corresponds to an input particle.) The missing nets can be generated by adding a so-called *reflexion* operator [6]; hence we refer to the action structure of all π -nets as RPIC, or reflexive PIC. Here we shall ignore these matters and stick to PIC, but all our points are relevant to PIC and RPIC alike.

In terms of the algebra we can now define reaction very succinctly. First we define

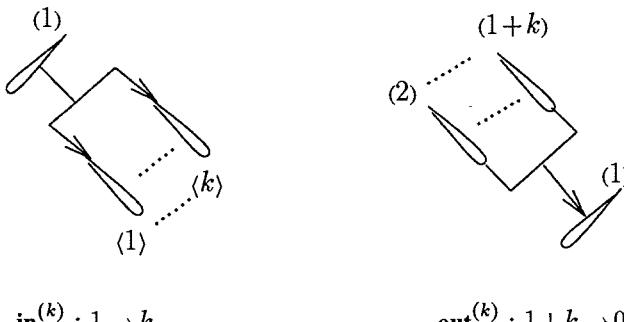
$$\begin{aligned}\mathbf{in}_u : 0 \rightarrow 1 &\stackrel{\text{def}}{=} \langle u \rangle \cdot \mathbf{in} \\ \mathbf{out}_u : 1 \rightarrow 0 &\stackrel{\text{def}}{=} (\langle u \rangle \otimes \mathbf{id}_1) \cdot \mathbf{out},\end{aligned}$$

which gives both ports the name tag u ; then the rule for single reaction is

$$\mathbf{out}_u \otimes \mathbf{in}_u \searrow^1 \mathbf{id}_1.$$

The one-step reaction relation \searrow^1 over π -nets is the smallest which satisfies this rule and is preserved by product, composition and abstraction. This corresponds exactly to our graphical description of reaction in Section 2.

Let us briefly return to the notion of multi-arcs, i.e. polyadic π -nets. These are obtained by generalizing the **in** and **out** generators to



³This has nothing to do with achieving infinite reaction sequences. Every π -nets as defined in Section 2 is reduced in size by a reaction, so infinite reaction is impossible. It will become possible when we introduce replication.

Then the reaction rule becomes

$$\text{out}_u^{(k)} \otimes \text{in}_u^{(k)} \searrow^1 \text{id}_k ,$$

which has the effect of simultaneously coalescing k pairs of source and target nodes. Part of the power of the boxes which we define below is that this polyadic reaction can be simulated by them, so (as was the case with original π -calculus) we lose nothing by taking the monadic form as basic.

Notation We shall freely use the following abbreviations:

$$\begin{aligned} ab &\text{ for } a \cdot b \\ \langle xy \rangle &\text{ for } \langle x \rangle \otimes \langle y \rangle \\ (x)a &\text{ for } \mathbf{ab}_x a \cdot (\omega \otimes \text{id}_n) \quad (a : m \rightarrow n) . \end{aligned}$$

For example, we have

$$\begin{aligned} (x)\langle ux \rangle \text{out} &= \text{out}_u \\ \langle ux \rangle \text{out} &= \langle x \rangle \cdot \text{out}_u . \end{aligned}$$

4 Adding boxes and replication

Recall the example we gave in Section 2 of a π -calculus process in original notation:

$$P = x(y).(\overline{y}\langle x \rangle \mid y(v).\overline{v}\langle z \rangle) ;$$

because the input prefix $x(y)$ guards what follows, the internal reaction at y cannot occur until an external reaction at x occurs. We cannot match this in PIC; a redex which consists of a pair of particles at the same port, without further instantiation of names, can *always* be reduced.

Recall also, from [5], how the polyadic π -calculus can be derived from the monadic form. This derivation requires that we define the polyadic prefix constructions $u(\vec{x}).P$ and $\overline{u}(\vec{x}).P$, where \vec{x} is any sequence of names. This is done as follows for sequences of length two, thus enabling a pair of names to be transmitted in a single reaction:

$$\begin{aligned} u(x_1 x_2).P &\stackrel{\text{def}}{=} u(w).w(x_1).w(x_2).P \\ \overline{u}\langle y_1 y_2 \rangle.Q &\stackrel{\text{def}}{=} (\nu w)\overline{u}\langle w \rangle.\overline{w}\langle y_1 \rangle.\overline{w}\langle y_2 \rangle.Q . \end{aligned}$$

Note that this uses both input and output monadic guards. Honda and Tokoro [3] made the remarkable discovery that in fact the input monadic guards are enough; with them we can get the power of output guards, and hence also of polyadic communication. Here, in essence, is their construction:

$$\begin{aligned} u(x_1 x_2).P &\stackrel{\text{def}}{=} u(w).(\nu v_1)(\overline{w}\langle v_1 \rangle \mid v_1(x_1).(\nu v_2)(\overline{w}\langle v_2 \rangle \mid v_2(x_2).P)) \\ \overline{u}\langle y_1 y_2 \rangle.Q &\stackrel{\text{def}}{=} (\nu w)\left(\overline{u}\langle w \rangle \mid w(v_1).(\overline{v_1}\langle y_1 \rangle \mid w(v_2).(\overline{v_2}\langle y_2 \rangle \mid Q))\right) . \end{aligned}$$

This has all the beauty of a zip-fastener; note how a parallel bar on one side is matched by a prefix on the other, and vice versa.

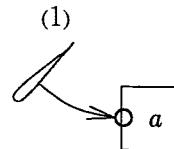
In action structures for π -calculus, it will be product (\otimes) and a construction called *boxing* which play this rôle. The construction of Honda and Tokoro justifies the *box* control construction of [7] as a sufficient extension of PIC for imposing sequence upon reactions. The remainder of this section is devoted to defining an extended form of π -net with boxing and replication. This is essentially the action calculus $\text{PIC}(\text{box}, \text{rep})$ which was defined in [7]. The new ingredient here is the graphical presentation; we shall use this in Section 5 to present the Honda-Tokoro construction.

We define the control operation *box*, which takes a single action as parameter, with the following arity rule:

$$\frac{a : 1 \rightarrow n}{\text{box } a : 1 \rightarrow n}.$$

The correspondence with the π -calculus is as follows: if a corresponds to the process abstraction $(x)P$ then $\langle u \rangle \cdot \text{box } a$ corresponds to the input prefix construction $u(x).P$. (Of course, the target arity n of a has no correspondent in the π -calculus.)

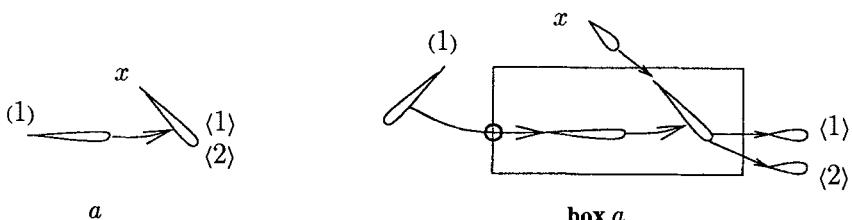
We indicate this construction in π -nets informally as follows:



$$\text{box } a : 1 \rightarrow n \quad (a : 1 \rightarrow n)$$

The ring indicates that the arc, which we shall call a *box arc*, impinges on a 's single import node, which loses its import tag.

In our formal constructions we require a bit more detail, to maintain the convention that each tag in a net occurs uniquely. Let us define a *region* of a net to be a part which is not crossed by a box boundary. We shall ensure that each tag occurs only in the outermost (unboxed) region of a net, as follows. We introduce a new kind of arc, called a *link*, drawn from a node head to a node tail. In forming *box a*, if any name tag or export tag occurs on a node of a then this node is linked to a new distinct external node, which receives the tag instead. Here is an example:

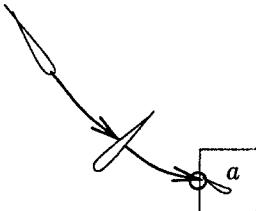


When boxes are nested, the linking for a name x forms a tree, making x everywhere

accessible. Intuitively, we can see that boxes exist to confine *arcs* (which are the ingredients of redexes), not nodes; so the links are just a graphical device to liberate nodes again.

In practice we need not bother always to add the new nodes and the links, since they can be uniquely supplied when missing. But with this formal treatment the operations of product, composition and abstraction remain exactly as they were described earlier. Note in particular that they will not coalesce any node in a box, and that they add no further links.

For boxes we have a new kind of redex, consisting of an arc entering and a box arc leaving the same node waist, thus:



To reduce such a redex we coalesce its source node with the import node of a , remove both arcs and the box surrounding a , and then coalesce any pair of linked nodes in the enlarged outer region, removing the link arcs.

Now, just as we did for **in**, we define

$$\text{box}_u a : 0 \rightarrow n \stackrel{\text{def}}{=} \langle u \rangle \text{box } a$$

and add a further reaction rule

$$\text{out}_u \otimes \text{box}_u a \searrow^1 a .$$

In passing, note that **in** becomes redundant in the presence of **box**, since it behaves exactly as **box id**. But the main point of concern here is as follows: if \searrow^1 is taken to be the smallest relation which satisfies this rule and is preserved by the action structure operations, the effect in terms of π -nets is that reaction cannot occur inside a box.

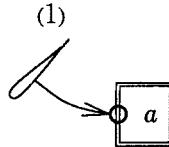
It is a routine matter to check that these operations and the reaction relation yield an action structure. It is very close to the action calculus PIC(box) defined in [7], but not identical; in Section 6 we look at the slight adjustment needed, and also some alternative reaction relations.

Let us now look briefly at replication. In [8] a form of replication, written $\pi * P$, was defined in which a copy of the process P is generated every time the prefix π is activated. In [5] a more general form $!P$ was defined, simply by imposing the axiom $!P = P \mid !P$; this amounts to declaring that an unbounded number of copies of P exist side-by-side without any need for activation.

Here we adopt the former approach, since it is closer to boxing. In fact, in the same spirit, we require the activating prefix to be an input action. This is the control operation **rep** which was defined in [7]. Its arity rule is

$$\frac{a : 1 \rightarrow 0}{\text{rep } a : 1 \rightarrow 0} ,$$

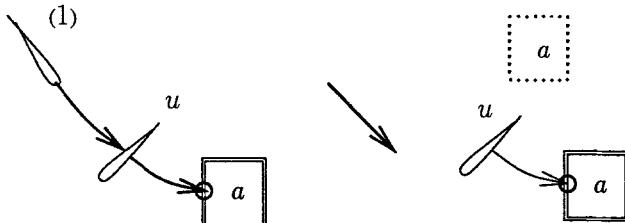
and we indicate the construction graphically as follows:



$$\mathbf{rep} \ a : 1 \rightarrow 0 \quad (a : 1 \rightarrow 0)$$

The reaction rule for **rep** is just as for **box**, except that the box is retained alongside the copy (which is why the target arity must be 0):

$$\begin{aligned} \mathbf{rep}_u a : 0 \rightarrow 0 &\stackrel{\text{def}}{=} (u)\mathbf{rep} \ a \\ \mathbf{out}_u \otimes \mathbf{rep}_u a &\searrow^1 a \otimes \mathbf{rep}_u a . \end{aligned}$$

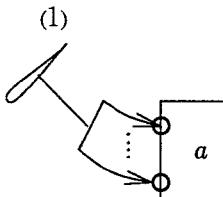


With this rule, we come close to the action calculus $\text{PIC}(\text{box}, \mathbf{rep})$ defined in [7]; a similar small adjustment is needed as for **box**. For the purposes of this paper we need say no more about **rep**, since it poses no more problems than **box** for graphical presentation.

5 Using boxes: the Honda-Tokoro construction

We shall now use the construction of Honda and Tokoro to simulate multi-arc reduction, using boxes. (We do not need replication.) We add nothing to their idea, but merely present it in the setting of π -nets.

Let us first generalise **box**, as we generalised **in**. We define the constructor $\mathbf{box}^{(k)}$ thus (so that $\mathbf{box} = \mathbf{box}^{(1)}$):

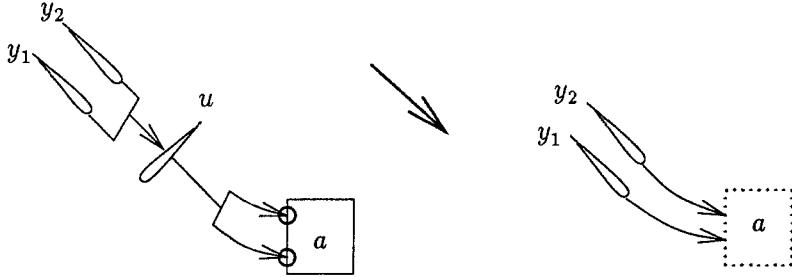


$$\mathbf{box}^{(k)} a : 1 \rightarrow n \quad (a : k \rightarrow n)$$

Correspondingly we generalise the reaction rule for **box**, for arbitrary $a : k \rightarrow n$:

$$\begin{array}{c} \mathbf{box}_u^{(k)} a : 0 \rightarrow n \quad \stackrel{\text{def}}{=} \quad \langle u \rangle \mathbf{box}^{(k)} a \\ \mathbf{out}_u^{(k)} \otimes \mathbf{box}_u^{(k)} a \quad \searrow^1 \quad a . \end{array}$$

For $k = 2$, if we tag the source nodes with y_1 and y_2 this reaction is just



Our first task then is to encode $\langle y_1 y_2 \rangle \mathbf{out}_u^{(2)}$ and $\mathbf{box}_u^{(2)} a$. We adapt the Honda-Tokoro expressions given at the beginning of this section, as follows (using square brackets for boxes, and staggered to show the structure):

$$\begin{aligned} \langle y_1 y_2 \rangle \mathbf{out}_u^{(2)} &\stackrel{\text{def}}{=} \nu(w) \left(\langle w \rangle \mathbf{out}_u \otimes \right. \\ &\quad \mathbf{box}_w [(v_1) \\ &\quad (\langle y_1 \rangle \mathbf{out}_{v_1} \otimes \\ &\quad \mathbf{box}_{v_1} [(v_2) \\ &\quad \langle y_2 \rangle \mathbf{out}_{v_2}])] \Big) \\ \mathbf{box}_u^{(2)} a &\stackrel{\text{def}}{=} \mathbf{box}_u \left[(w) \right. \\ &\quad \nu(v_1) (\langle v_1 \rangle \mathbf{out}_w \otimes \\ &\quad \mathbf{box}_{v_1} [(x_1) \\ &\quad \nu(v_2) (\langle v_2 \rangle \mathbf{out}_w \otimes \\ &\quad \mathbf{box}_{v_2} [(x_2) (\langle x_1 x_2 \rangle a)])]) \Big] . \end{aligned}$$

Now take the product

$$b_0 = \langle y_1 y_2 \rangle \mathbf{out}_u^{(2)} \otimes \mathbf{box}_u^{(2)} a ;$$

we must show that $b_0 \searrow \langle y_1 y_2 \rangle a$. We show b_0 as a net in Figure 1. Some annotation has been added to help comparison with the expressions; formally, the only alphabetic symbols in the net are y_1 , y_2 and u (apart from the schematic variable a).

The reduction sequence

$$b_0 \searrow^1 b_1 \searrow^1 \cdots \searrow^1 b_5 \sim \langle y_1 y_2 \rangle a$$

is shown in Figure 2. At each stage one box is exploded. Note that there are two links; they are the arcs which cross a box boundary, *other than* box arcs which are

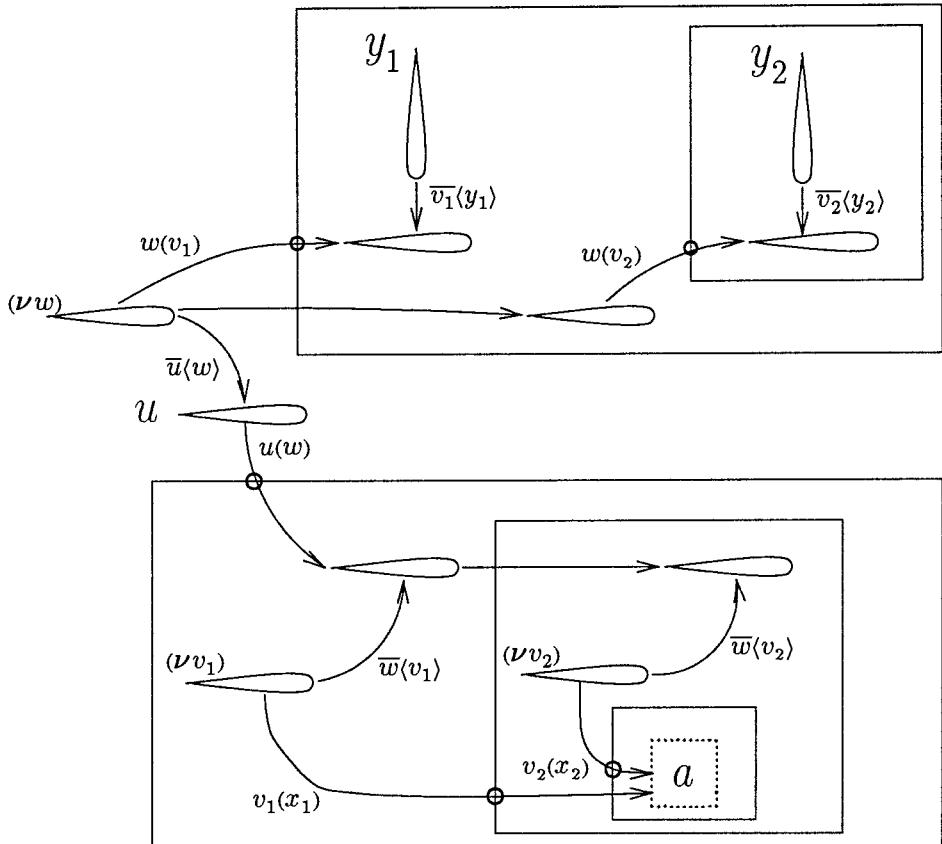


Figure 1: $b_0 = \langle y_1 y_2 \rangle \mathbf{out}_u^{(2)} \otimes \mathbf{box}_u^{(2)} a$

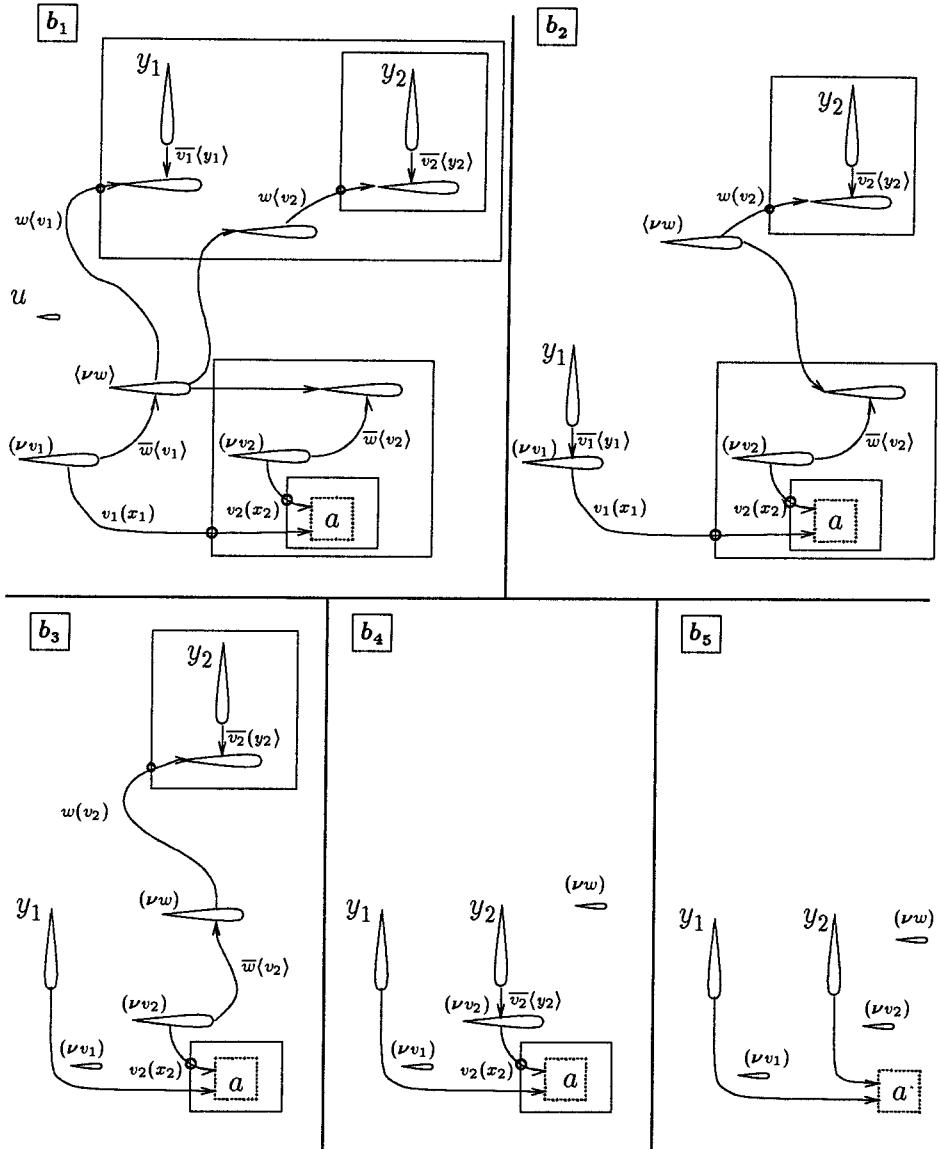


Figure 2: Reduction of b_0 : $b_0 \searrow^1 b_1 \searrow^1 b_2 \searrow^1 b_3 \searrow^1 b_4 \searrow^1 b_5 \sim \langle y_1 y_2 \rangle a$

distinguished by a ring. It is exactly the control of these links which imposes the proper sequence on reduction to ensure that y_1 is bound to x_1 and y_2 to x_2 .

Essentially we have $b_5 = \langle y_1 y_2 \rangle a$ as required; the only difference is that b_5 has three arcless, tagless nodes as garbage. This garbage is highly inert, and persists harmlessly through all operations and reductions. (An algebraic aside: this garbage is neatly removed by imposing the identity $\nu \cdot \omega = \text{id}_0$ upon the action structure.)

Does this reduction sequence entitle us to claim that we have encoded $\langle y_1 y_2 \rangle \text{out}_u^{(2)}$ properly? Not fully; for its existence still allows that some b_i may have other possible reductions, or may take part in them when placed in a suitable context. If that were so, the encoding would be invalid. However, it is not so. One can convince oneself by inspecting the nets that every arc shown could never take part in any other reduction. To justify this claim rigorously requires an adaptation of the notions of *reachability* (of arcs) and *incident*, discussed in [6].

6 Variations

The reaction relation we have chosen for boxes is probably the smallest which is reasonable. But there is a series of variations which are weaker, in the sense that they allow earlier coalescence of nodes and may therefore permit certain reactions to occur earlier. The Honda-Tokoro construction works in all of them; this is some evidence that each of them imposes “sufficient” sequencing. This appears to be due to the common feature of all the variations: they all require the two arcs of a redex to lie within the same region.

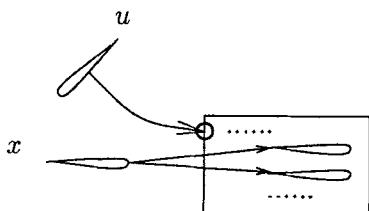
Link immigration We begin by pointing out how the action structure defined above differs from the action calculus $\text{PIC}(\text{box})$ given in [7]. Consider

$$a = \langle xx \rangle \cdot (yz)\text{box}_u[\dots y \dots z \dots];$$

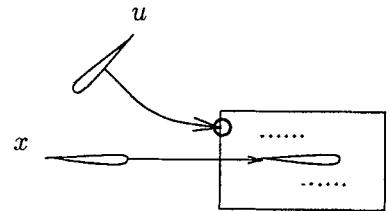
if y, z do not occur elsewhere in the box, in $\text{PIC}(\text{box})$ this is equal to

$$b = \text{box}_u[\dots x \dots x \dots],$$

i.e. the instantiation of y and z to x may permeate the box. But a and b correspond to different nets:



a



b

Let us call a link *inward* if it points into a box. Then we achieve the effect of instantiation permeating a box if we equate each pair of nets like a and b , i.e. whenever two inward links have the same source node then we may coalesce their target nodes. This induces an equivalence relation upon π -nets (actually a congruence with respect to all the constructions); when we divide by this congruence, I conjecture that the action structure becomes isomorphic to $\text{PIC}(\text{box})$.

If this coalescence is not made, the situation amounts to the enforced delay of a substitution. This strongly suggests the notion of *explicit substitution*, studied by Abadi et al [1] for the λ -calculus. Indeed, by extending PIC to allow substitution particles like (x/y) (pronounced “ x for y ”, where x is free and y bound) within the body of an action, we would hope to model delayed substitutions in an action calculus.

Reaction within a box In the Honda-Tokoro construction the essential use of boxing was to prevent an arc outside a box from reacting with one inside. This purpose is still perfectly achieved even if we allow reaction within a box, or more exactly within any region. This is expressed not by changing the reaction rules

$$\begin{array}{lcl} \text{out}_u \otimes \text{in}_u & \searrow^1 & \text{id}_1 \\ \text{out}_u \otimes \text{box}_u a & \searrow^1 & a \\ \text{out}_u \otimes \text{rep}_u a & \searrow^1 & a \otimes \text{rep}_u a \end{array}$$

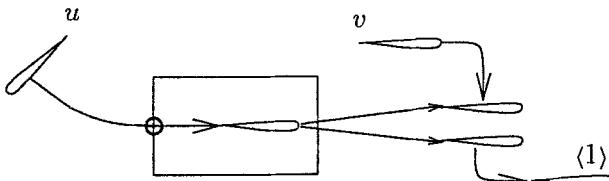
but simply by requiring that they may be applied in any context – i.e. that the reaction relation be preserved by the box and replication constructions as well as by the action structure operations. Of course one may choose to allow reaction within a box but forbid it within a replication (perhaps on the grounds that such reactions count for more, since their effect will be multiplied indefinitely).

In the context of this more liberal regime, link immigration also takes on a greater significance because it will enable earlier reaction within a box.

Link emigration There is a phenomenon dual to the immigration of links. Consider

$$a = \text{box}_u[(x)(xx)] \cdot (yz)(\langle v \rangle \text{out}_y \otimes \text{in}_z) ;$$

is there a reaction? Apparently not; if we look at the net we can see that the reaction will wait until the box is exploded.



This is in agreement with $\text{PIC}(\text{box})$ as defined in [7] (in contrast with the case for link immigration). There is no obvious reason why such an implication for coalescing two

nodes should not be allowed to permeate from the inside to the outside of a box. If reaction within boxes is also allowed (as discussed above), then this is a way to allow any coalescences which arise from internal reaction to be felt externally; the box still prevents reaction *between* its interior and exterior.

How can this outward permeation be achieved? In π -nets we can proceed dually to the immigration case. Let us call a link *outward* if it points out of a box. Then we get what we want by equating each pair of nets like the following:



i.e. whenever two outward links have the same source node then we may coalesce their target nodes. Again, this induces a congruence relation upon π -nets.

How can we match this congruence algebraically? We would hope to match it by imposing an equational axiom in addition to those of [7]. At first sight, there is an intriguing possibility which may be correct; it is to choose an appropriate *effect structure* E (see [4]) for π -nets, and to impose the axiom

$$\text{box}(a \cdot e) = \text{box } a \cdot e \quad (e \in E).$$

Certainly $(x)(xx)$ is a member of the natural effect structure, so we would have for the above example

$$\begin{aligned} a &= \text{box}_u[\text{id}_1] \cdot (x)(xx) \cdot (yz)(\langle v \rangle \text{out}_y \otimes \text{in}_z) \\ &= \text{box}_u[\text{id}_1] \cdot (x)(\langle v \rangle \text{out}_x \otimes \text{in}_x) \end{aligned}$$

which has a reaction. The ramifications of this idea are interesting; the richer the effect structure, the greater is the influence which reaction within a box may exert upon the environment – without losing the purpose of the box. One should also recall that since effects are inert, to export them from the box cannot reduce the possibility of internal reactions.

This idea remains largely unexplored, and deserves further research.

7 Conclusion

The aim of this informal paper has been to explore the graphical presentation of the various action structures which enrich the π -calculus.

By exploring π -nets with boxes we have illustrated the power of boxing; we have also found a surprising degree of freedom in its precise meaning. Of course the Honda-Tokoro construction was known, and some of the various of the boxing regimes discussed in the previous section were vaguely familiar; so π -nets were not strictly necessary for some of the observations which we have made. But I had no idea about the phenomenon of *link emigration* before looking at π -nets and trying to formalise them. The nets therefore seem to be at least a good auxiliary tool of investigation, provided that they are not allowed to prevent a more abstract treatment when it becomes

appropriate. We have also seen that there is good hope for algebraic characterisation of the various boxing regimes.

Little has been said here about the family of *action calculi*, of which PIC and PIC(box, rep) and its variants are members. But the graphical presentation is by no means restricted to these members of the family. It may be fruitful to identify a class of action calculi which yield naturally to graphical representation; this may be of value not only theoretically, but also for the implementation of such an action calculus considered as a programming language.

References

- [1] Abadi, M., Cardelli, L., Curien, P-L. and Lévy, J-J., *Explicit substitutions*, *Journal of Functional Programming* 1, 4 (October 1991), 375–416.
- [2] Berry, G. and Boudol, G., *The chemical abstract machine*, *Journal of Theoretical Computer Science*, Vol 96, pp217–248, 1992.
- [3] Honda, K. and Tokoro, M., *An object calculus for asynchronous communication*, Proc. European Conference on object-oriented programming, Lecture Notes in Computer Science, Vol 512, Springer, pp133–147, 1991.
- [4] Milner, R., *Action structures*, Research Report LFCS-92-249, Laboratory for Foundations of Computer Science, Computer Science Department, Edinburgh University, 1992.
- [5] Milner, R., *The polyadic π -calculus: a tutorial*, in **Logic and Algebra of Specification**, ed. F.L. Bauer, W. Brauer and H. Schwichtenberg, Springer Verlag, 1993, pp203–246.
- [6] Milner, R., *Action structures for the π -calculus*, Research Report ECS-LFCS-93-264, Laboratory for Foundations of Computer Science, Computer Science Department, Edinburgh University, 1992.
- [7] Milner, R., *Action calculi I: axioms and applications*, Computer Science Department, Edinburgh University, 1993. Also appeared as *Action calculi, or syntactic action structures*, Proceedings of MFCS '93, Lecture Notes in Computer Science, Springer, 1993.
- [8] Milner, R., Parrow, J. and Walker D., *A calculus of mobile processes, Parts I and II*, *Journal of Information and Computation*, Vol 100, pp1–40 and pp41–77, 1992.
- [9] Parrow, J., *Interaction diagrams*, SICS Research Report R93:06, 1993. To appear in proceedings of REX'93 Workshop, Lecture Notes in Computer Science, Springer Verlag.

Local Type Reconstruction by means of Symbolic Fixed Point Iteration

Torben Amtoft

internet: tamtoft@daimi.aau.dk

DAIMI, Aarhus University

Ny Munkegade, DK-8000 Århus C, Denmark

Abstract. We convert, via a version that uses constraints, a type inference system for strictness analysis into an algorithm which given an expression finds the set of possible typings. Although this set in general does not possess a minimal element, it can be represented compactly by means of *symbolic expressions* in *normal form* – such expressions have the property that once values for the constraint variables with *negative polarity* have been supplied it is straight-forward to compute the minimal values for the constraint variables with *positive polarity*. The normalization process works *on the fly*, i.e. by a leaf-to-root traversal of the inference tree.

1 Background and Motivation

Recently much interest has been devoted to the formulation of program analysis in terms of inference systems, as opposed to e.g. abstract interpretation (for the relationship between those methods see e.g. [Jen91]). This approach is appealing since it separates the question “what is done?” from the question “how is it done?”. Of course, the latter issue (that is, implementation of the inference system) has to be dealt with, and a very popular method (often inspired by [Hen91]) is to (re)formulate the inference system in terms of *constraints* and then come up with an algorithm for solving these.

In this paper we shall continue this trend, the type system in question being one for *strictness analysis* (the system has also been presented in [Amt93a]). The characteristic features of our approach are:

- for constraints we define a notion of *normal form* which distinguishes between constraint variables according to their polarity (i.e. whether they occur in co/contravariant position in the type).
- the constraints are normalized *on the fly*, that is during a leaf-to-root traversal of the inference tree (as opposed to first collecting them all and then solve them).
- during the normalization process, some *approximations* are made – without losing precision, however, since the approximation only concerns suboptimal solutions.
- the normalization process involves *symbolic fixed point iteration*.

2 Introduction

We shall be working with the typed and extended λ -calculus. An expression is either a constant c ; a variable x ; an abstraction $\lambda x.e$; an application $e_1 e_2$; a conditional if $e_1 e_2 e_3$ or a recursive definition $\text{rec } f \ e$. The set of *underlying* types will be denoted \mathcal{U} ; such a type is either a base type (`Int`, `Bool`, `Unit` etc.) or a function type $u_1 \rightarrow u_2$. `Base` will denote some base type.

Strictness Analysis: The analysis we want to perform is *strictness analysis*, that is the task of detecting whether a function needs its argument(s). Recent years have seen many approaches to strictness analysis, most based on abstract interpretation – the starting point being the work of Mycroft [Myc80] which was extended to higher order functions by Burn, Hankin and Abramsky [BHA86].

Kuo and Mishra [KM89] presented a type system where types t are formed from 0 (denoting non-termination), 1 (denoting non-termination or termination, i.e. any term) and $t_1 \rightarrow t_2$. Accordingly, if it is possible to assign a function the type $0 \rightarrow 0$ we know that the function is strict. Wright has proposed alternative type systems [Wri91] where the idea is to annotate the *arrows*, according to whether a function is strict or not.

Strictness Types: The type system used in this paper is the one of Wright, except that we use subtyping instead of polymorphism. Accordingly we define the set of *strictness types*, \mathcal{T} , as follows: a strictness type t is either a base type `Base` or a *strict* function type $t_1 \rightarrow_0 t_2$ (denoting that we *know* that the function is strict) or a *general* function type $t_1 \rightarrow_1 t_2$ (denoting that we do not know whether the function is strict).

We shall impose an ordering \leq on strictness types, defined by stipulating that $t_1 \rightarrow_b t_2 \leq t'_1 \rightarrow_{b'} t'_2$ iff $t'_1 \leq t_1$, $b \leq b'$ and $t_2 \leq t'_2$, and by stipulating that `Int` \leq `Int` etc. The intuitive meaning of $t \leq t'$ is that t is more informative than t' ; for instance it is more informative to know that a function is of type `Int` \rightarrow_0 `Int` than to know that it is of type `Int` \rightarrow_1 `Int`. Similarly, it is more informative to know that a function is of type $(\text{Int} \rightarrow_1 \text{Int}) \rightarrow_0 (\text{Int} \rightarrow_0 \text{Int})$ (it maps *arbitrary* functions into strict functions) than to know that it is of type $(\text{Int} \rightarrow_0 \text{Int}) \rightarrow_0 (\text{Int} \rightarrow_0 \text{Int})$ (it maps strict functions into strict functions).

An Inference System: In Fig. 1 we present the inference system the implementation of which is the topic of this paper. An application of the inference system can be found in [Amt93a], where it is shown how one by means of an inference tree can produce a (provably correct) translation from call-by-name into call-by-value which inserts fewer “suspensions” than the naive translation.

A judgment in the system takes the form $\Gamma \vdash e : t, W$ where

- Γ is an environment, represented as a list (i.e. of form $((x_1 : t_1) \dots (x_n : t_n))$), assigning strictness types to variables;
- e is an expression such that if x is a free variable of e then $\Gamma(x)$ is defined;
- t is a strictness type;

- W maps the domain of Γ into $\{0, 1\}$. It may be helpful to think of W as follows: if $W(x) = 0$ then x is needed in order to evaluate e to “head normal form”. If $\Gamma = ((x_1 : t_1) \dots (x_n : t_n))$ and if $W(x_i) = b_i$ we shall often write $W = (b_1 \dots b_n)$.

A brief explanation of the inference system: the first inference rule (the “subsumption rule”) is non-structural and expresses the ability to forget information: if an expression has type t and needs the variables in W , it also has any more imprecise type and will also need any subset of W . The application of this rule might for instance be needed in order to assign the same type to the two branches in a conditional. The rule for constants employs a function CT with the property that for all constants c it holds that $CT(c)$ is a *strict iterated base type*; that is either Base or of type $\text{Base} \rightarrow_0 t$ with t a strict iterated base type. Note in the rule for variables that in order to evaluate x it is necessary to evaluate x but no other variables are needed. The rule for abstractions says that if x is among the variables needed by e then $\lambda x.e$ can be assigned a strict type (\rightarrow_0), otherwise not. The rule for applications says that the variables needed to evaluate e_1 are also needed to evaluate $e_1 e_2$; and if e_1 is strict then the variables needed to evaluate e_2 will also be needed to evaluate $e_1 e_2$. The rule for conditionals says that if a variable is needed to evaluate the test then it is also needed to evaluate the whole expression; and also if a variable is needed in order to evaluate *both* branches it will be needed to evaluate the whole expression. The rule for recursion says that if the assumption that f has type t suffices for proving that the body e has type t then the construction $\text{rec } f \ e$ also has type t ; and that if a variable different from f is needed to evaluate e then this variable is also needed to evaluate $\text{rec } f \ e$.

Fig. 1 is a *specification* of an analysis (and in [Amt93a] the specification is proven “correct” wrt. an operational semantics); in particular no clue is given on how to *implement* the analysis, that is how to construct a type for a given expression (and its subexpressions). As we want to focus upon the strictness annotations, we shall assume that the underlying types have been given in advance.

Principal Types: For type inference (type reconstruction) in general, an expression can be assigned many different types – fortunately, however, it is often possible to find a “principal type” such that all other types can be “derived” from this type. We shall now investigate to which extent there exists a “principal typing” in our setting. As a (running) example, we consider the *twice* function defined by $\lambda f.\lambda x.f(f(x))$. If the underlying type of *twice* has been fixed to $(\text{Int} \rightarrow \text{Int}) \rightarrow (\text{Int} \rightarrow \text{Int})$, a valid strictness type must be of form $(\text{Int} \rightarrow b_1 \text{Int}) \rightarrow b_2 (\text{Int} \rightarrow b_3 \text{Int})$ where the b_i ’s are *strictness variables*, that is variables which range over $\{0, 1\}$. It is not too difficult to see that all assignments to the b_i ’s are valid except the typings of form $(\text{Int} \rightarrow \text{Int}) \rightarrow b_2 (\text{Int} \rightarrow_0 \text{Int})$ (as *twice*, given a non-strict function, doesn’t produce a strict function); in other words we have the (sufficient and necessary) constraint $b_3 \geq b_1$.

$$\begin{array}{l}
\frac{\Gamma \vdash e : t, W}{\Gamma \vdash e : t', W'}, \text{ if } t' \geq t, W' \geq W \\
\Gamma \vdash c : CT(c), \vec{1} \\
(\Gamma_1, (x : t), \Gamma_2) \vdash x : t, (\vec{1}, 0, \vec{1}) \\
\frac{((x : t_1), \Gamma) \vdash e : t, (b, W)}{\Gamma \vdash \lambda x. e : t_1 \rightarrow_b t, W} \\
\frac{\Gamma \vdash e_1 : t_2 \rightarrow_b t_1, W_1 \quad \Gamma \vdash e_2 : t_2, W_2}{\Gamma \vdash e_1 e_2 : t_1, W} \\
\text{if } W(x) = W_1(x) \sqcap (b \sqcup W_2(x)) \text{ for all } x \\
\frac{\Gamma \vdash e_1 : \text{Bool}, W_1 \quad \Gamma \vdash e_2 : t, W_2 \quad \Gamma \vdash e_3 : t, W_3}{\Gamma \vdash (\text{if } e_1 \ e_2 \ e_3) : t, W} \\
\text{if } W(x) = W_1(x) \sqcap (W_2(x) \sqcup W_3(x)) \text{ for all } x \\
\frac{((f : t), \Gamma) \vdash e : t, (b, W)}{\Gamma \vdash (\text{rec } f \ e) : t, W}
\end{array}$$

Fig. 1. An inference system for strictness types.

Polarity: In the *twice* example, the polarity of b_1 is *negative* (as it occurs nested within an odd number of contravariant positions) and the polarity of b_2 and b_3 is *positive*. So the constraint produced ($b_3 \geq b_1$) yields a principal typing in the following sense: given a value for the negative strictness variable, it is trivial to find the set of possible values for the positive strictness variables. As we shall see later in this paper, this is a general phenomenon: if \vec{b}^+ are the positive strictness variables and \vec{b}^- the negative strictness variables of the overall type (i.e. the type of the “whole” expression), it is possible to generate a “vector” constraint which is of form $\vec{b}^+ \geq g(\vec{b}^-)$ with g a monotone function. (In the *twice* example we have $(b_2, b_3) \geq (0, b_1)$, that is $g(b_1) = (0, b_1)$.)

Strictness Expressions. Monotone functions will be represented by *strictness expressions*, which are built from strictness variables, 0, 1, \sqcap and \sqcup . A strictness expression s gives rise to a monotone function g with domain the free variables of s (and a little thought reveals that all monotone functions on finite domains can be represented by strictness expressions).

Approximating Solutions: We are not satisfied by only knowing the annotations of the arrows occurring in the *overall* type; we would like to know the typing of the *subexpressions* also (as this is needed by e.g. the translation algorithm in [Amt93a]). As an example, consider the term $((\text{twice } id) 7)$ (with *twice* as before and with $id = \lambda x. x$). The overall type of this term is *Int*; as we have seen the type of *twice* is of form $(\text{Int} \rightarrow_{b_1} \text{Int}) \rightarrow_{b_2} (\text{Int} \rightarrow_{b_3} \text{Int})$; and the type of *id* is of form

$\text{Int} \rightarrow_{b_4} \text{Int}$. As before there is a constraint $b_3 \geq b_1$, and as the type of a function must match the type of its argument we also have a constraint $b_1 = b_4$. Just as the values of the positive variables of the overall type could be found from the negative variables of the overall type, we would like to be able to find the values of the “interior” strictness variables (i.e. b_1, b_2, b_3, b_4) from the negative variables of the overall type (of which there are none in the above case). A first attempt at achieving this would be to look for constants c_i such that the solutions are given by the constraints $b_i \geq c_i$. This, however, is not possible: we must clearly have $c_1 = c_4 = 0$ but the constraints $b_1 \geq 0, b_4 \geq 0$ are not sufficient for a solution as they forget the requirement that $b_1 = b_4$. In order to represent those requirements succinctly we devise a symbol \geq which is “sandwiched” between $=$ and \geq in a sense made precise by the following two definitions:

Definition 1. Let N be an extended constraint system (i.e. possibly containing \geq). Let ϕ be a mapping from the strictness variables of N into $\{0, 1\}$. We say that ϕ is a *strong solution* to N iff ϕ is a solution to the system resulting from replacing all \geq ’s in N by $=$; and we say that ϕ is a *weak solution* to N iff ϕ is a solution to the system resulting from replacing all \geq ’s in N by \geq .

Notice that for constraint systems not containing \geq , strong and weak solutions coincide. Also note that a strong solution is also a weak solution.

Definition 2. Let N_1 and N_2 be extended constraint systems. We say that N_2 approximates N_1 , to be written $N_1 \triangleleft N_2$, iff

- all strong solutions to N_2 are strong solutions to N_1 , and
- all weak solutions to N_1 are weak solutions to N_2 .

Clearly \triangleleft is reflexive and transitive; and referring back to the example it is easy to see that $\{b_1 = b_4\} \triangleleft \{b_1 \geq 0, b_4 \geq 0\}$. The right hand side contains less information than the left hand side, but as we can assume that we aim for minimal solutions this does not really matter.

Normalizing Constraints: In Sect. 3 we shall see that Fig. 1 can be rewritten into a system with judgments of form $\Gamma \vdash e : t, W, C$ where C is a set of constraints. The strictness variables occurring in C can be divided into two groups: those occurring in t or Γ , to be denoted \vec{b}_1^+ and \vec{b}_1^- , and those which do not occur in t or Γ (i.e. the *interior* variables, which occur further up in the proof tree), to be denoted \vec{b}_0 (we do not consider polarity here). As indicated above there exists a *normalization* algorithm which produces an extended constraint system N with the following property:

- $C \triangleleft N$;
- N is of form $\{\vec{b}_1^+ \geq \vec{s}_1, \vec{b}_0 \geq \vec{s}_0\}$, with \vec{s}_0 and \vec{s}_1 being strictness expressions whose (free) variables belong to \vec{b}_1^- .

The algorithm is defined inductively in the inference tree – for details, see Sect. 5.

Manipulating Symbolic Expressions: From the above it follows that one can view type reconstruction as a function $T(e, \vec{b}^-)$ which given an expression together with values for the negative strictness variables of its type returns the (minimal) values for the positive strictness variables (and the interior variables). A key point of our approach is that the normalization algorithm, which manipulates symbolic expressions, essentially does *partial evaluation* of T wrt. e – all computation dependent on e is done once only resulting in a piece of code which rapidly can be applied to many different values of \vec{b}^- .

This paradigm seems rather widely applicable; previous examples include [Con91] (for a first-order strictness analysis) and [Ros79] (for a flow-analysis of an imperative language).

Fixed Point Solving: An important technique to be applied during normalization is *fixed point iteration*. To see an example of this, consider the function

$$\lambda g. \lambda h. \text{rec } f \lambda x. \text{if } (g x) (h x) (f (- 1 x)) .$$

Let us *outline* how to type this function: we can assume that g has type $\text{Int} \rightarrow_{b_g} \text{Bool}$ and h has type $\text{Int} \rightarrow_{b_h} \text{Int}$. Likewise we can assume that the formal parameter f has type $\text{Int} \rightarrow_{b_f} \text{Int}$ and that the body of f has type $\text{Int} \rightarrow_{b_f} \text{Int}$. It is not too hard to see that we must have the constraint $b_f \geq b_g \sqcap (b_h \sqcup b_f)$ and due to the rule for recursion we must also have $b = b_f$, that is we have the constraint

$$b_f \geq b_g \sqcap (b_h \sqcup b_f) \tag{1}$$

The negative variables of the overall type are b_g and b_h ; we are thus left with the task of expressing b_f in terms of these two strictness variables, i.e. to find an expression s where only b_g and b_h occur free such that the right hand side of (1) can be rewritten into s . We shall find s as the limit of the chain s_0, s_1, s_2, \dots where

- $s_0 = 0$;
- $s_1 = b_g \sqcap (b_h \sqcup s_0) = b_g \sqcap (b_h \sqcup 0) = b_g \sqcap b_h$;
- $s_2 = b_g \sqcap (b_h \sqcup s_1) = b_g \sqcap (b_h \sqcup (b_g \sqcap b_h)) = (b_g \sqcap b_h) \sqcup (b_g \sqcap b_g \sqcap b_h) = b_g \sqcap b_h$.

We see that the chain reaches its limit after one iteration, which was to be expected: since the lattice $\{0 \sqsubseteq 1\}$ has height one, for each value of b_g and b_h a fixed point will be reached in one step – we shall therefore reach a *representation* of all those fixed points after one step. Thus we can replace (1) by the constraint

$$b_f \geq b_g \sqcap b_h$$

and have thus found out that f will be strict if either g or h is strict – this should come as no surprise. The reason for writing \geq instead of \geq is that even though the right hand side represents the *least* fixed point, in general it does not hold that any value above this point is a fixed point – however, in this simple case where only one variable is present it would make no difference whether we write \geq or \geq .

For a formalization of the reasoning performed (implicitly) above, see Lemma 8 (and its proof).

One of the reasons for preferring type inference to abstract interpretation is that the latter approach usually involves (expensive) fixed point computation (whereas the former employs unification). Therefore it may seem suspicious that fixed point computation has crept into our type inference framework. On the other hand, it turns out that for a given expression the number of fixed point iterations we have to perform will be bounded by the sum of the sizes of the types of its subexpressions. Hence we can assume that for “typical” programs the cost of fixed point iteration will not be exorbitant.

Structure of Paper: The rest of the paper is organized as follows: in Sect. 3 we transform the inference system from Fig. 1 into one based on constraints; in Sect. 4 we list some tools to be used during the normalization process; and in Sect. 5 we give a detailed account of how to combine these tools into an algorithm for solving the constraints. Section 6 concludes.

3 Rewriting the Inference System

The first step towards getting a system more suitable for implementation is to “inline” the subsumption rule into (some of) the other rules¹. The result is depicted in Fig. 2; it is not difficult to see that this system is equivalent to the one in Fig. 1 in the sense that the same judgments can be derived.

$$\begin{array}{l}
 \Gamma \vdash c : t, \vec{1} \text{ if } t \geq CT(c) \\
 \\
 (\Gamma_1, (x : t), \Gamma_2) \vdash x : t', (\vec{1}, b, \vec{1}) \text{ if } t' \geq t, b \geq 0 \\
 \\
 \frac{((x : t_1), \Gamma) \vdash e : t, (b, W)}{\Gamma \vdash \lambda x. e : t_1 \rightarrow_b t, W} \\
 \\
 \frac{\Gamma \vdash e_1 : t_2 \rightarrow_b t_1, W_1 \quad \Gamma \vdash e_2 : t_2, W_2}{\frac{\Gamma \vdash e_1 e_2 : t_1, W}{\text{if } W(x) \geq W_1(x) \sqcap (b \sqcup W_2(x)) \text{ for all } x}} \\
 \\
 \frac{\Gamma \vdash e_1 : \text{Bool}, W_1 \quad \Gamma \vdash e_2 : t_2, W_2 \quad \Gamma \vdash e_3 : t_3, W_3}{\frac{\Gamma \vdash (\text{if } e_1 e_2 e_3) : t, W}{\text{if } t \geq t_2, t \geq t_3, W(x) \geq W_1(x) \sqcap (W_2(x) \sqcup W_3(x)) \text{ for all } x}} \\
 \\
 \frac{((f : t), \Gamma) \vdash e : t, (b, W)}{\Gamma \vdash (\text{rec } f e) : t', W} \text{ if } t' \geq t
 \end{array}$$

Fig. 2. The result of inlining the subsumption rule.

¹ This inlining of the non-structural rules into the structural rules is a very common technique for getting an implementable system; another (more complex) example can be found in [SNN92].

The next step is to make the annotations on arrows more explicit; at the same time distinguishing between positive/negative polarity. To this end we introduce some notation: if u is a (standard) type in \mathcal{U} , and if \vec{b}^+ and \vec{b}^- are vectors of 0 or 1's, then $u[\vec{b}^+, \vec{b}^-]$ denotes u where all positive arrows are marked (from left to right) as indicated by \vec{b}^+ and where all negative arrows are marked (from left to right) as indicated by \vec{b}^- . More formally, we have

- $\text{Base}[((),())] = \text{Base};$
- $(u_1 \rightarrow u_2)[(\vec{b}_1^+, b^+, \vec{b}_2^+), (\vec{b}_1^-, b^-, \vec{b}_2^-)] = u_1[\vec{b}_1^-, \vec{b}_1^+] \rightarrow_{b^+} u_2[\vec{b}_2^+, \vec{b}_2^-].$

Example: with $u = ((\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int}) \rightarrow ((\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int})$, we have

$$u[(b_1, b_2, b_3), (b_4, b_5)] = ((\text{Int} \rightarrow_{b_1} \text{Int}) \rightarrow_{b_4} \text{Int}) \rightarrow_{b_2} ((\text{Int} \rightarrow_{b_5} \text{Int}) \rightarrow_{b_3} \text{Int}) .$$

If $\Gamma = ((x_1 : u_1) \dots (x_n : u_n))$ then

$$\Gamma[(\vec{b}_1^-, \dots, \vec{b}_n^-), (\vec{b}_1^+, \dots, \vec{b}_n^+)] = ((x_1 : u_1[\vec{b}_1^-, \vec{b}_1^+]), \dots, (x_n : u_n[\vec{b}_n^-, \vec{b}_n^+])) .$$

Fact 3. $u[\vec{b}_1^+, \vec{b}_1^-] \leq u[\vec{b}_2^+, \vec{b}_2^-]$ iff $\vec{b}_1^+ \leq \vec{b}_2^+$ and $\vec{b}_2^- \leq \vec{b}_1^-$ (pointwise).

Using this notation the system from Fig. 2 rewrites into the system depicted² in Fig. 3. A remark about polarity: the turnstile \vdash acts like an \rightarrow , so if $t = \Gamma(x)$ then positive positions in t will be considered as appearing negatively in the judgment, and vice versa. On the other hand, something appearing in the range of W is considered as being in positive position in the judgment. It is important to notice that the polarity of a strictness variable is an unambiguous notion, i.e. it is always the same in the premise of a rule as in the conclusion. We shall consistently use the convention that the polarity of a variable can be read from its superscript (as e.g. in \vec{b}_1^+).

It is straightforward to transform the system in Fig. 3 into one using *constraints* among the strictness variables, that is one with judgments of form $\Gamma \vdash e : t, W, C$ with C a set of constraints: just turn the side conditions into constraints. The resulting system is depicted in Fig. 4.

Recall our assumption that the underlying types have been given in advance. Then the inference system in Fig. 4 in an obvious way gives rise to a deterministic algorithm collecting a set of constraints. We are thus left with the task of *solving* those constraints. As indicated in Sect. 2 our approach will be to show how to *normalize* the constraints inductively in the proof tree (“on the fly”), thus demonstrating that the solutions have a particular form.

Before embarking on the normalization process, it will be convenient to describe some of the tools to be used.

² For space reasons we shall employ the convention that e.g. “ $\Gamma \vdash e_1 : t_1, W_1$ and $e_2 : t_2, W_2$ ” means “ $\Gamma \vdash e_1 : t_1, W_1$ and $\Gamma \vdash e_2 : t_2, W_2$ ”. Also, with some abuse of notation, we shall write CT for the function (defined on the set of constants) into \mathcal{U} which first applies the “old” CT and then “removes the annotations from the arrows”.

$$\begin{array}{c}
\frac{\Gamma[\vec{b}^-, \vec{b}^+] \vdash c : CT(c)[\vec{b}_1^+, ()], \vec{b}_2^+}{\text{if } \vec{b}_1^+ \geq \vec{0}, \vec{b}_2^+ \geq \vec{1}} \\
\\
\frac{(\Gamma_1[\vec{b}_1^-, \vec{b}_1^+], (x : u[\vec{b}_2^-, \vec{b}_2^+]), \Gamma_2[\vec{b}_3^-, \vec{b}_3^+]) \vdash x : u[\vec{b}_4^+, \vec{b}_4^-], (\vec{b}_5^+, \vec{b}_6^+, \vec{b}_7^+) \text{ if } \vec{b}_4^+ \geq \vec{b}_2^-, \vec{b}_2^+ \geq \vec{b}_4^-, \vec{b}_5^+ \geq 1, \vec{b}_6^+ \geq 0, \vec{b}_7^+ \geq 1}{((x : u_1[\vec{b}_1^-, \vec{b}_1^+]), \Gamma[\vec{b}^-, \vec{b}^+]) \vdash e : u[\vec{b}_2^+, \vec{b}_2^-], (b_3^+, \vec{b}_4^+)} \\
\\
\frac{\Gamma[\vec{b}^-, \vec{b}^+] \vdash \lambda x.e : u_1[\vec{b}_1^-, \vec{b}_1^+] \rightarrow_{\vec{b}_3^+} u[\vec{b}_2^+, \vec{b}_2^-], \vec{b}_4^+}{\Gamma[\vec{b}^-, \vec{b}^+] \vdash e_1 : u_2[\vec{b}_2^-, \vec{b}_2^+] \rightarrow_{\vec{b}_3^+} u_1[\vec{b}_4^+, \vec{b}_4^-], \vec{b}_5^+ \text{ and } e_2 : u_2[\vec{b}_6^+, \vec{b}_6^-], \vec{b}_7^+} \\
\\
\frac{\Gamma[\vec{b}^-, \vec{b}^+] \vdash e_1 e_2 : u_1[\vec{b}_4^+, \vec{b}_4^-], \vec{b}_8^+ \text{ if } \vec{b}_6^+ = \vec{b}_2^-, \vec{b}_2^+ = \vec{b}_6^-, \vec{b}_8^+ \geq \vec{b}_5^+ \sqcap (\vec{b}_3^+ \sqcup \vec{b}_7^+)}{\Gamma[\vec{b}^-, \vec{b}^+] \vdash e_1 : \text{Bool}, \vec{b}_3^+ \text{ and } e_2 : u[\vec{b}_4^+, \vec{b}_4^-], \vec{b}_5^+ \text{ and } e_3 : u[\vec{b}_6^+, \vec{b}_6^-], \vec{b}_7^+ \text{ if } \vec{b}_8^+ \geq \vec{b}_4^+, \vec{b}_4^- \geq \vec{b}_8^-, \vec{b}_8^+ \geq \vec{b}_6^+, \vec{b}_6^- \geq \vec{b}_8^-, \vec{b}_9^+ \geq \vec{b}_3^+ \sqcap (\vec{b}_5^+ \sqcup \vec{b}_7^+)} \\
\\
\frac{((f : u[\vec{b}_1^-, \vec{b}_1^+]), \Gamma[\vec{b}^-, \vec{b}^+]) \vdash e : u[\vec{b}_2^+, \vec{b}_2^-], (b_3^+, \vec{b}_4^+)}{\Gamma[\vec{b}^-, \vec{b}^+] \vdash (\text{rec } f \ e) : u[\vec{b}_5^+, \vec{b}_5^-], \vec{b}_4^+ \text{ if } \vec{b}_5^+ \geq \vec{b}_2^+, \vec{b}_2^- \geq \vec{b}_5^-, \vec{b}_2^+ = \vec{b}_1^-, \vec{b}_2^- = \vec{b}_1^+}
\end{array}$$

Fig. 3. Annotations on arrows made explicit.

4 Some Transformation Rules

Given a constraint system N , there are several ways to come up with a system N' such that $N \triangleleft N'$. Below we list some of these methods:

Fact 4. Suppose N contains the constraints $\vec{b} \geq \vec{s}_1$, $\vec{b} \geq \vec{s}_2$. Then these can be replaced by the constraint $\vec{b} \geq \vec{s}_1 \sqcup \vec{s}_2$.

Fact 5. Suppose N contains the constraint $\vec{b} \geq \vec{s}$ or the constraint $\vec{b} = \vec{s}$. Then this can be replaced by $\vec{b} \geq \vec{s}$.

Lemma 6. Suppose N contains the constraints $\vec{b}_1 \geq g_1(\vec{b}_2)$ and $\vec{b}_2 \geq g_2(\vec{b}_3)$. Then the former constraint can be replaced by the constraint $\vec{b}_1 \geq g_1(g_2(\vec{b}_3))$, yielding a new constraint system N' .

(In other words, if we have the constraints $\vec{b} \geq \vec{s}$ and $\vec{b}_i \geq \vec{s}_i$, it is safe to replace the former constraint by $\vec{b} \geq \vec{s}[\vec{s}_i / \vec{b}_i]$.)

Proof. Let a strong solution to N' be given. Wrt. this solution, we have $\vec{b}_2 = g_2(\vec{b}_3)$, $\vec{b}_1 \geq g_1(g_2(\vec{b}_3))$ and hence also $\vec{b}_1 \geq g_1(\vec{b}_2)$ – thus this solution is also a strong solution to N .

$$\begin{aligned}
& \Gamma[\vec{b}^-, \vec{b}^+] \vdash c : CT(c)[\vec{b}_1^+, ()], \vec{b}_2^+, \\
& \quad \{\vec{b}_1^+ \geq \vec{0}, \vec{b}_2^+ \geq \vec{1}\} \\
& (\Gamma_1[\vec{b}_1^-, \vec{b}_1^+], (x : u[\vec{b}_2^-, \vec{b}_2^+]), \Gamma_2[\vec{b}_3^-, \vec{b}_3^+]) \vdash x : u[\vec{b}_4^+, \vec{b}_4^-], (\vec{b}_5^+, \vec{b}_6^+, \vec{b}_7^+), \\
& \quad \{\vec{b}_4^+ \geq \vec{b}_2^-, \vec{b}_2^+ \geq \vec{b}_4^-, \vec{b}_5^+ \geq 1, \vec{b}_6^+ \geq 0, \vec{b}_7^+ \geq 1\} \\
& \frac{((x : u_1[\vec{b}_1^-, \vec{b}_1^+]), \Gamma[\vec{b}^-, \vec{b}^+]) \vdash e : u[\vec{b}_2^+, \vec{b}_2^-], (\vec{b}_3^+, \vec{b}_4^+), C}{\Gamma[\vec{b}^-, \vec{b}^+] \vdash \lambda x. e : u_1[\vec{b}_1^-, \vec{b}_1^+] \rightarrow_{\vec{b}_3^+} u[\vec{b}_2^+, \vec{b}_2^-], \vec{b}_4^+, C} \\
& \frac{\Gamma[\vec{b}^-, \vec{b}^+] \vdash e_1 : u_2[\vec{b}_2^-, \vec{b}_2^+] \rightarrow_{\vec{b}_3^+} u_1[\vec{b}_4^+, \vec{b}_4^-], \vec{b}_5^+, C_1 \text{ and } e_2 : u_2[\vec{b}_6^+, \vec{b}_6^-], \vec{b}_7^+, C_2}{\Gamma[\vec{b}^-, \vec{b}^+] \vdash e_1 e_2 : u_1[\vec{b}_4^+, \vec{b}_4^-], \vec{b}_8^+, C_1 \cup C_2 \cup C} \\
& \quad \text{where } C = \{\vec{b}_6^+ = \vec{b}_2^-, \vec{b}_2^+ = \vec{b}_4^-, \vec{b}_8^+ \geq \vec{b}_5^+ \sqcap (\vec{b}_3^+ \sqcup \vec{b}_7^+)\} \\
& \frac{\Gamma[\vec{b}^-, \vec{b}^+] \vdash e_1 : \text{Bool}, \vec{b}_3^+, C_1 \text{ and } e_2 : u[\vec{b}_4^+, \vec{b}_4^-], \vec{b}_5^+, C_2 \text{ and } e_3 : u[\vec{b}_6^+, \vec{b}_6^-], \vec{b}_7^+, C_3}{\Gamma[\vec{b}^-, \vec{b}^+] \vdash (\text{if } e_1 \ e_2 \ e_3) : u[\vec{b}_8^+, \vec{b}_8^-], \vec{b}_9^+, C_1 \cup C_2 \cup C_3 \cup C} \\
& \quad \text{where } C = \{\vec{b}_8^+ \geq \vec{b}_4^+, \vec{b}_4^- \geq \vec{b}_8^-, \vec{b}_8^+ \geq \vec{b}_6^+, \vec{b}_6^- \geq \vec{b}_8^-, \vec{b}_9^+ \geq \vec{b}_3^+ \sqcap (\vec{b}_5^+ \sqcup \vec{b}_7^+)\} \\
& \frac{((f : u[\vec{b}_1^-, \vec{b}_1^+]), \Gamma[\vec{b}^-, \vec{b}^+]) \vdash e : u[\vec{b}_2^+, \vec{b}_2^-], (\vec{b}_3^+, \vec{b}_4^+), C}{\Gamma[\vec{b}^-, \vec{b}^+] \vdash (\text{rec } f \ e) : u[\vec{b}_5^+, \vec{b}_5^-], \vec{b}_6^+, C \cup C'} \\
& \quad \text{where } C' = \{\vec{b}_5^+ \geq \vec{b}_2^+, \vec{b}_2^- \geq \vec{b}_5^-, \vec{b}_2^+ = \vec{b}_1^-, \vec{b}_2^- = \vec{b}_1^+\}
\end{aligned}$$

Fig. 4. An inference system collecting constraints.

Now let a weak solution to N be given. Wrt. this solution, we have $\vec{b}_1 \geq g_1(\vec{b}_2)$ and $\vec{b}_2 \geq g_2(\vec{b}_3)$. Due to the monotonicity of g_1 , we also have $\vec{b}_1 \geq g_1(g_2(\vec{b}_3))$ showing that this solution is also a weak solution to N' . \square

Lemma 7. Suppose N contains the constraints $\vec{b}_1 \geq g_1(\vec{b}_2)$ and $\vec{b}_2 \geq g_2(\vec{b}_3)$. Then the former constraint can be replaced by the constraint $\vec{b}_1 \geq g_1(g_2(\vec{b}_3))$, yielding a new constraint system N' .

(In other words, if we have the constraints $\vec{b} \leqq \vec{s}$ and $\vec{b}_i \leqq \vec{s}_i$, it is safe to replace the former constraint by $\vec{b} \leqq \vec{s}[\vec{s}_i / \vec{b}_i]$.)

Proof. Let a strong solution to N' be given. Wrt. this solution, we have $\vec{b}_2 = g_2(\vec{b}_3)$, $\vec{b}_1 = g_1(g_2(\vec{b}_3))$ and hence also $\vec{b}_1 = g_1(\vec{b}_2)$ – thus this solution is also a strong solution to N .

Now let a weak solution to N be given. Wrt. this solution, we have $\vec{b}_1 \geq g_1(\vec{b}_2)$ and $\vec{b}_2 \geq g_2(\vec{b}_3)$. Due to the monotonicity of g_1 , we also have $\vec{b}_1 \geq g_1(g_2(\vec{b}_3))$ showing that this solution is also a weak solution to N' . \square

Lemma 8. Suppose N contains the constraint $\vec{b}_1 \geq g(\vec{b}_1, \vec{b}_2)$. Then this can be replaced by the constraint $\vec{b}_1 \geq g'(\vec{b}_2)$ (yielding N'), where

$$g'(\vec{b}_2) = \sqcup_k h^k(\vec{0}) \text{ with } h(\vec{b}) = g(\vec{b}, \vec{b}_2)$$

(this just amounts to Tarski's theorem – notice that it will actually suffice with $|\vec{b}_1|$ iterations).

Proof. First suppose that we have a strong solution to N' , i.e. $\vec{b}_1 = g'(\vec{b}_2)$. Since $\vec{b}_1 = \sqcup_k h^k(\vec{0})$, standard reasoning on the monotone and hence (as everything is finite) continuous function h tells us that $\vec{b}_1 = h(\vec{b}_1)$, i.e. $\vec{b}_1 = g(\vec{b}_1, \vec{b}_2)$. This shows that we have a strong solution to N .

Now suppose that we have a weak solution to N , i.e. $\vec{b}_1 \geq g(\vec{b}_1, \vec{b}_2)$. In order to show that this also is a weak solution to N' , we must show that $\vec{b}_1 \geq g'(\vec{b}_2)$. This can be done by showing that $\vec{b}_1 \geq \vec{b}$ implies $\vec{b}_1 \geq h(\vec{b})$. But if $\vec{b}_1 \geq \vec{b}$ we have $\vec{b}_1 \geq g(\vec{b}_1, \vec{b}_2) \geq g(\vec{b}, \vec{b}_2) = h(\vec{b})$.

It is easy to see that g' is monotone. \square

5 The Normalization Process

We shall examine the various constructs: for constants and variables the normalization process is trivial, as the constraints generated are of the required form. Neither does the rule for abstractions cause any trouble, since no new constraints are generated and since a strictness variable appears in the premise of the rule iff it appears in the conclusion (and with the same polarity). Now let us focus upon the remaining constructs, where for space reasons we omit conditionals (can be found in [Amt93b]).

Normalizing the Rule for Application. Recall the rule

$$\frac{\Gamma[\vec{b}^-, \vec{b}^+] \vdash e_1 : u_2[\vec{b}_2^-, \vec{b}_2^+] \rightarrow_{\vec{b}_3^+} u_1[\vec{b}_4^+, \vec{b}_4^-], \vec{b}_5^+, C_1 \text{ and } e_2 : u_2[\vec{b}_6^+, \vec{b}_6^-], \vec{b}_7^+, C_2}{\Gamma[\vec{b}^-, \vec{b}^+] \vdash e_1 e_2 : u_1[\vec{b}_4^+, \vec{b}_4^-], \vec{b}_8^+, C_1 \cup C_2 \cup C}$$

where $C = \{\vec{b}_6^+ = \vec{b}_2^-, \vec{b}_2^+ = \vec{b}_6^-, \vec{b}_8^+ \geq \vec{b}_5^+ \sqcap (\vec{b}_3^+ \sqcup \vec{b}_7^+)\}$.

Let \vec{b}_0 be the “extra” strictness variables of C_1 and \vec{b}_1 the extra strictness variables of C_2 . Then we are inductively entitled to assume that there exist N_1, N_2 with $C_1 \triangleleft N_1$, $C_2 \triangleleft N_2$ such that N_1 takes the form

$$\vec{b}^+ \geq \vec{s}_a \vec{b}_2^+ \geq \vec{s}_2 \vec{b}_3^+ \geq \vec{s}_3 \vec{b}_4^+ \geq \vec{s}_4 \vec{b}_5^+ \geq \vec{s}_5 \vec{b}_0 \geq \vec{s}_0$$

(where the free variables of the strictness expressions above belong to $\{\vec{b}^-, \vec{b}_2^-, \vec{b}_4^-\}$ and such that N_2 takes the form

$$\vec{b}^+ \geq \vec{s}_b \vec{b}_6^+ \geq \vec{s}_6 \vec{b}_7^+ \geq \vec{s}_7 \vec{b}_1 \geq \vec{s}_1$$

(where the free variables of the strictness expressions above belong to $\{\vec{b}^-, \vec{b}_6^-\}$.)

Clearly $C_1 \cup C_2 \cup C \triangleleft N_1 \cup N_2 \cup C$. We shall now manipulate $N_1 \cup N_2 \cup C$ with the aim of getting something of the desired form.

The first step is to use Fact 4 to replace the two inequalities for \vec{b}^+ by one, and at the same time exploit that $\vec{b}_2^+ = \vec{b}_6^-$ and $\vec{b}_6^+ = \vec{b}_2^-$. As a result, we arrive at

$$\begin{aligned}\vec{b}^+ &\geq \vec{s}_a \sqcup \vec{s}_b \quad \vec{b}_6^- \geq \vec{s}_2 \quad b_3^+ \geq s_3 \quad \vec{b}_4^+ \geq \vec{s}_4 \\ \vec{b}_1^+ &\geq \vec{s}_5 \quad \vec{b}_0 \geq \vec{s}_0 \quad \vec{b}_7^- \geq \vec{s}_6 \quad \vec{b}_7^+ \geq \vec{s}_7 \\ \vec{b}_1 &\geq \vec{s}_1 \quad \vec{b}_6^+ = \vec{b}_2^- \quad \vec{b}_2^+ = \vec{b}_6^- \quad \vec{b}_8^+ \geq \vec{b}_5^+ \sqcap (b_3^+ \sqcup \vec{b}_7^+)\end{aligned}$$

We now focus upon the pair of constraints

$$(\vec{b}_6^-, \vec{b}_2^-) \geq (\vec{s}_2, \vec{s}_6) .$$

According to Lemma 8, these can be replaced by the constraints

$$(\vec{b}_6^-, \vec{b}_2^-) \geq (\vec{S}_2, \vec{S}_6)$$

where (\vec{S}_2, \vec{S}_6) is given as the “limit” of the chain with elements $(\vec{s}_{2n}, \vec{s}_{6n})$ given by

$$\begin{aligned}(\vec{s}_{20}, \vec{s}_{60}) &= \vec{0} \\ (\vec{s}_{2(n+1)}, \vec{s}_{6(n+1)}) &= (\vec{s}_2, \vec{s}_6)[(\vec{s}_{2n}, \vec{s}_{6n}) / (\vec{b}_6^-, \vec{b}_2^-)]\end{aligned}$$

This limit can be found as the k 'th element, where $k = |(\vec{b}_2^-, \vec{b}_6^-)|$.

As \vec{s}_2 does not contain \vec{b}_6^- and \vec{s}_6 does not contain \vec{b}_2^- , the above can be simplified into

$$\begin{aligned}\vec{s}_{20} &= \vec{0} \quad \vec{s}_{2(n+1)} = \vec{s}_2[\vec{s}_{6n}/\vec{b}_2^-] \\ \vec{s}_{60} &= \vec{0} \quad \vec{s}_{6(n+1)} = \vec{s}_6[\vec{s}_{2n}/\vec{b}_6^-]\end{aligned}$$

Our next step is to substitute in the new constraints for \vec{b}_2^- and \vec{b}_6^- , using Lemma 6 and Lemma 7. We arrive at (after also having used Fact 5)

$$\begin{aligned}\vec{b}^+ &\geq \vec{s}_a[\vec{S}_6/\vec{b}_2^-] \sqcup \vec{s}_b[\vec{S}_2/\vec{b}_6^-] \quad \vec{b}_6^- \geq \vec{S}_2 \quad b_3^+ \geq s_3[\vec{S}_6/\vec{b}_2^-] \\ \vec{b}_4^+ &\geq \vec{s}_4[\vec{S}_6/\vec{b}_2^-] \quad \vec{b}_4^+ \geq \vec{s}_5[\vec{S}_6/\vec{b}_2^-] \quad \vec{b}_0 \geq \vec{s}_0[\vec{S}_6/\vec{b}_2^-] \\ \vec{b}_2^- &\geq \vec{S}_6 \quad \vec{b}_7^+ \geq \vec{s}_7[\vec{S}_2/\vec{b}_6^-] \quad \vec{b}_1 \geq \vec{s}_1[\vec{S}_2/\vec{b}_6^-] \\ \vec{b}_6^+ &\geq \vec{S}_6 \quad \vec{b}_2^+ \geq \vec{S}_2 \quad \vec{b}_8^+ \geq \vec{b}_5^+ \sqcap (b_3^+ \sqcup \vec{b}_7^+)\end{aligned}$$

Finally we use Lemma 6 on the constraint for \vec{b}_8^+ , arriving at

$$\begin{aligned}\vec{b}^+ &\geq \vec{s}_a[\vec{S}_6/\vec{b}_2^-] \sqcup \vec{s}_b[\vec{S}_2/\vec{b}_6^-] \quad \vec{b}_6^- \geq \vec{S}_2 \quad b_3^+ \geq s_3[\vec{S}_6/\vec{b}_2^-] \\ \vec{b}_4^+ &\geq \vec{s}_4[\vec{S}_6/\vec{b}_2^-] \quad \vec{b}_5^+ \geq \vec{s}_5[\vec{S}_6/\vec{b}_2^-] \quad \vec{b}_0 \geq \vec{s}_0[\vec{S}_6/\vec{b}_2^-] \\ \vec{b}_2^- &\geq \vec{S}_6 \quad \vec{b}_7^+ \geq \vec{s}_7[\vec{S}_2/\vec{b}_6^-] \quad \vec{b}_1 \geq \vec{s}_1[\vec{S}_2/\vec{b}_6^-] \\ \vec{b}_6^+ &\geq \vec{S}_6 \quad \vec{b}_2^+ \geq \vec{S}_2 \quad \vec{b}_8^+ \geq \vec{s}_5[\vec{S}_6/\vec{b}_2^-] \sqcap (s_3[\vec{S}_6/\vec{b}_2^-] \sqcup \vec{s}_7[\vec{S}_2/\vec{b}_6^-])\end{aligned}$$

This is of the desired form, as it is quite easy to check that the only strictness variables occurring in the expressions on the right hand sides are those occurring in \vec{b}^- and in \vec{b}_4^- .

Normalizing the Rule for Recursion. Recall the rule

$$\frac{((f : u[\vec{b}_1^-, \vec{b}_1^+]), \Gamma[\vec{b}^-, \vec{b}^+]) \vdash e : u[\vec{b}_2^+, \vec{b}_2^-], (b_3^+, \vec{b}_4^+), C}{\Gamma[\vec{b}^-, \vec{b}^+] \vdash \text{rec } f e : u[\vec{b}_5^+, \vec{b}_5^-], \vec{b}_4^+, C \cup C'}$$

where $C' = \{\vec{b}_1^- = \vec{b}_2^+, \vec{b}_2^- = \vec{b}_1^+, \vec{b}_5^+ \geq \vec{b}_2^+, \vec{b}_2^- \geq \vec{b}_5^-\}$.

Let \vec{b}_0 be the “extra” strictness variables of C . We are inductively entitled to assume that there exist N with $C \triangleleft N$ such that N takes the form

$$\vec{b}^+ \geq \vec{s} \vec{b}_1^+ \geq \vec{s}_1 \vec{b}_2^+ \geq \vec{s}_2 \vec{b}_3^+ \geq \vec{s}_3 \vec{b}_4^+ \geq \vec{s}_4 \vec{b}_0 \geq \vec{s}_0$$

(where the free variables of the strictness expressions above belong to $(\vec{b}^-, \vec{b}_1^-, \vec{b}_2^-)$.)

Clearly $C \cup C' \triangleleft N \cup C'$. We shall now manipulate $N \cup C'$ with the aim of getting something of the desired form.

The first step is to exploit that $\vec{b}_2^- = \vec{b}_1^+$ and $\vec{b}_1^- = \vec{b}_2^+$, and afterwards exploit Fact 4 to replace the two inequalities for \vec{b}_2^- by one. We arrive at

$$\begin{aligned} \vec{b}^+ &\geq \vec{s} \quad \vec{b}_2^- \geq \vec{s}_1 \sqcup \vec{b}_5^- \quad \vec{b}_1^- \geq \vec{s}_2 \\ \vec{b}_3^+ &\geq \vec{s}_3 \quad \vec{b}_4^+ \geq \vec{s}_4 \quad \vec{b}_0 \geq \vec{s}_0 \\ \vec{b}_1^- &= \vec{b}_2^+ \quad \vec{b}_2^- = \vec{b}_1^+ \quad \vec{b}_5^+ \geq \vec{b}_2^+ \end{aligned}$$

We now focus upon the pair of constraints

$$(\vec{b}_2^-, \vec{b}_1^-) \geq (\vec{s}_1 \sqcup \vec{b}_5^-, \vec{s}_2) .$$

According to Lemma 8, these can be replaced by the constraints

$$(\vec{b}_2^-, \vec{b}_1^-) \geq (\vec{S}_1, \vec{S}_2)$$

where (\vec{S}_1, \vec{S}_2) is given as the “limit” of the chain with elements $(\vec{s}_{1n}, \vec{s}_{2n})$ given by

$$\begin{aligned} (\vec{s}_{10}, \vec{s}_{20}) &= \vec{0} \\ (\vec{s}_{1(n+1)}, \vec{s}_{2(n+1)}) &= (\vec{s}_1 \sqcup \vec{b}_5^-, \vec{s}_2)[(\vec{s}_{1n}, \vec{s}_{2n}) / (\vec{b}_2^-, \vec{b}_1^-)] \end{aligned}$$

This limit can be found as the k 'th element, where $k = |(\vec{b}_1^-, \vec{b}_2^-)|$.

Our next step is to substitute in the new constraints for \vec{b}_1^- and \vec{b}_2^- , using Lemma 6 and Lemma 7. We arrive at (after also having used Fact 5 to replace $=$ by \geq)

$$\begin{array}{lll} \vec{b}^+ \geq \vec{s}[(\vec{S}_1, \vec{S}_2) / (\vec{b}_2^-, \vec{b}_1^-)] & \vec{b}_2^- \geq \vec{S}_1 & \vec{b}_1^- \geq \vec{S}_2 \\ \vec{b}_3^+ \geq \vec{s}_3[(\vec{S}_1, \vec{S}_2) / (\vec{b}_2^-, \vec{b}_1^-)] & \vec{b}_4^+ \geq \vec{s}_4[(\vec{S}_1, \vec{S}_2) / (\vec{b}_2^-, \vec{b}_1^-)] & \vec{b}_0 \geq \vec{s}_0[(\vec{S}_1, \vec{S}_2) / (\vec{b}_2^-, \vec{b}_1^-)] \\ \vec{b}_2^+ \geq \vec{b}_1^- & \vec{b}_1^+ \geq \vec{b}_2^- & \vec{b}_5^+ \geq \vec{b}_2^+ \end{array}$$

Finally we use Lemma 7 on the inequalities for \vec{b}_2^+ and \vec{b}_1^+ , and subsequently use Lemma 6 on the inequality for \vec{b}_5^+ . At the same time we replace some \geq by \geqq , and arrive at

$$\begin{array}{lll} \vec{b}^+ \geq \vec{s}_1[(\vec{S}_1, \vec{S}_2)/(\vec{b}_2^-, \vec{b}_1^-)] & \vec{b}_2^- \geq \vec{S}_1 & \vec{b}_1^- \geqq \vec{S}_2 \\ \vec{b}_3^+ \geqq \vec{s}_3[(\vec{S}_1, \vec{S}_2)/(\vec{b}_2^-, \vec{b}_1^-)] & \vec{b}_4^+ \geq \vec{s}_4[(\vec{S}_1, \vec{S}_2)/(\vec{b}_2^-, \vec{b}_1^-)] & \vec{b}_0 \geqq \vec{s}_0[(\vec{S}_1, \vec{S}_2)/(\vec{b}_2^-, \vec{b}_1^-)] \\ \vec{b}_2^+ \geqq \vec{S}_2 & \vec{b}_1^+ \geqq \vec{S}_1 & \vec{b}_5^+ \geq \vec{S}_2 \end{array}$$

This is of the desired form, as it is quite easy to check that the only strictness variables occurring in the expressions on the right hand sides are those occurring in \vec{b}^- and in \vec{b}_5^- .

6 Concluding Remarks

We have shown how to convert an inference system for strictness analysis into an algorithm that works by manipulating symbolic expressions. It would be interesting to see if the method described in this paper could be used on other kinds of inference systems.

A work bearing strong similarities to ours is described in [CJ93]. Here a type inference system for binding time analysis is presented. The idea is to annotate the function arrows with the *program* variables that *perhaps* are needed to evaluate the function (by replacing “perhaps” by “surely” one could obtain a strictness analysis); hence this analysis is stronger than the one presented in this paper. The price to be paid for the increased precision is that the type reconstruction algorithm presented in [CJ93] requires the user to supply the *full* type for bound variables – in our framework, this essentially means that the user has to supply the values of the negative variables (and some of the positive ones too). A remark is made in the paper that to automatize this seems to require some sort of second-order unification (undecidable); at the cost of performing fixed point iteration we achieve the desired effect (but, it should be emphasized, for a less precise analysis).

The type inference algorithm has been implemented in Miranda³. The user interface is as follows: first the user writes a λ -expression, and provides the underlying type of the bound variables; then the system returns an inference tree where all arrows are annotated with strictness variables, together with a normalized set of constraints among those variables. Next the user provides the values of the constraint variables occurring negatively in the overall type; and finally the system produces an inference tree where all subexpressions are assigned the least possible strictness type. For a full documentation see [Amt93b].

Future work includes investigating the complexity of the algorithm just developed. This involves choosing an appropriate input size parameter (could be the size of the expression, the size of its type, the maximal size of a subexpression’s type etc.). Also we have to think more carefully about a suitable representation of strictness expressions (in the implementation, a very naive representation was chosen).

³ Miranda is a trademark of Research Software Limited.

Acknowledgements: The author is supported by the DART-project funded by the Danish Research Councils and by the LOMAPS-project funded by ESPRIT. The work reported here evolved from numerous discussions with Hanne Riis Nielson and Flemming Nielson. Also thanks to Jens Palsberg for useful feedback.

References

- [Amt93a] Torben Amtoft. Minimal thunkification. In *3rd International Workshop on Static Analysis (WSA '93), September 1993, Padova, Italy*, number 724 in LNCS, pages 218–229. Springer-Verlag, 1993.
- [Amt93b] Torben Amtoft. Strictness types: An inference algorithm and an application. Technical Report PB-448, DAIMI, University of Aarhus, Denmark, August 1993.
- [BHA86] Geoffrey L. Burn, Chris Hankin, and Samson Abramsky. Strictness analysis for higher-order functions. *Science of Computer Programming*, 7:249–278, 1986.
- [CJ93] Charles Consel and Pierre Jouvelot. Separate polyvariant binding-time analysis. Technical Report CS/E 93-006, Oregon Graduate Institute, Department of Computer Science and Engineering, 1993.
- [Con91] Charles Consel. Fast strictness analysis via symbolic fixpoint iteration. Technical Report YALEU/DCS/RR-867, Yale University, September 1991.
- [Hen91] Fritz Henglein. Efficient type inference for higher-order binding-time analysis. In John Hughes, editor, *International Conference on Functional Programming Languages and Computer Architecture*, number 523 in LNCS, pages 448–472. Springer-Verlag, August 1991.
- [Jen91] Thomas P. Jensen. Strictness analysis in logical form. In John Hughes, editor, *International Conference on Functional Programming Languages and Computer Architecture*, number 523 in LNCS, pages 352–366. Springer-Verlag, August 1991.
- [KM89] Tsung-Min Kuo and Prateek Mishra. Strictness analysis: A new perspective based on type inference. In *International Conference on Functional Programming Languages and Computer Architecture '89*, pages 260–272. ACM Press, September 1989.
- [Myc80] Alan Mycroft. The theory of transforming call-by-need to call-by-value. In B. Robinet, editor, *International Symposium on Programming, Paris*, number 83 in LNCS, pages 269–281. Springer-Verlag, April 1980.
- [Ros79] Barry K. Rosen. Data flow analysis for procedural languages. *Journal of the ACM*, 26(2):322–344, April 1979.
- [SNN92] Kirsten Lackner Solberg, Hanne Riis Nielson, and Flemming Nielson. Inference systems for binding time analysis. In M. Billaud et al., editor, *Analyse statique, Bordeaux 92 (WSA '92)*, pages 247–254, September 1992.
- [Wri91] David A. Wright. A new technique for strictness analysis. In *TAPSOFT '91*, number 494 in LNCS, pages 235–258. Springer-Verlag, April 1991.

An Asynchronous Process Algebra with Multiple Clocks

Henrik Reif Andersen and Michael Mendler

Technical University of Denmark, Department of Computer Science,
Building 344, DK-2800 Lyngby, Denmark.
E-mail: {hra,mvm}@id.dtu.dk.

Abstract. In this paper we introduce a novel approach to the specification of real-time behaviour with process algebras. In contrast to the usual pattern, involving a fixed, measurable, and global notion of time, we suggest to represent real-time constraints indirectly through uninterpreted *clocks* enforcing broadcast synchronization between processes. Our approach advocates the use of *asynchronous* process algebras, which admit the faithful representation of nondeterministic and distributed computations.

Technically, we present a non-trivial extension of the *Calculus of Communicating Systems CCS* [Mil89a] by *multiple clocks* with associated *timeout* and *clock ignore* operators. We illustrate the flexibility of the proposed process algebra, called *PMC*, by presenting examples of rather different nature. The timeout operators generalize the timeout of ATP [NS90] to multiple clocks. The main technical contribution is a complete axiomatization of strong bisimulation equivalence for a class of finite-state processes and a complete axiomatization of observation congruence for finite processes.

1 Introduction

According to consolidating tradition in timed process algebras a real-time system is perceived to operate under the regime of a global time parameter constraining the occurrence of actions [NS91b]. Time has algebraic structure, typically a totally ordered commutative monoid, to express quantitative timing constraints. The semantics of a timed process then is given as a transition system enriched by quantitative timing information such as the absolute duration of actions or their time of occurrence.

This paper puts forward yet another process algebra; why bother? Most of the salient approaches, such as [MT90, Wan90, Lia91, NS91a, Klu91, SDJ⁺91], primarily aim at describing completely the global real-time behaviour of timed systems in a fairly realistic fashion. The means for abstracting from real time is restricted to the choice of the time domain; for instance, instead of working with real numbers one may decide to go for rational or discrete time. We believe that these approaches are often overly realistic with disadvantages for both the specification and the modelling of real-time systems. Firstly, for specifying a timed process, complete quantitative information about the intended timing behaviour of the implementation is required. This includes not only the specification-relevant, i.e. safety-critical timing, but also specification-irrelevant timing parameters which, so we believe, constitute the majority in practice. Being forced to include a lot of irrelevant timing information, which

can be left well up to the implementation, is unfortunate as this is unnecessarily cutting down the design space, perhaps even preventing the designer from finding a reasonable implementation at all. Secondly, many real-time process algebras require to give exact numbers for the duration of actions, such as “3.141 time units to enter a valid login response”. Examples are Timed-ACP as described in [Klu91], Timed-CSP [SDJ⁺91], ATPD [NS91a], or [Wan90]. But exact delays are in general very difficult to implement due to uncontrollable fabrication parameters, operating conditions such as circuit temperature or external events. At best we can hope to implement delay intervals. A process algebra using delay intervals rather than exact time was proposed by Liang [Lia91]. Such an algebra, however, suffers even more from being cluttered up with irrelevant timing information. Another process algebra with interval durations is CIPA [AM93a]. A disadvantage of time intervals are the severe problems they cause for simulation, in particular where time is dense: It is not feasible faithfully to simulate time intervals for the purpose of timing validation.

In this paper we propose a rather abstract approach to the specification and modelling of real-time systems that captures the nature of timing constraints through the use of *multiple clocks*. Clocks enforce global synchronization of actions without compromising the abstractness of time by referring to a concrete time domain which is globally fixed once and for all.

The concept of time underlying the use of clocks is abstract, qualitative, and local. Firstly, it is *abstract* since it does not prejudice any particular way of realizing a clock. We are free to interpret a clock as the ticking of a global real-time watch measuring absolute process time, as the system clock of a synchronous processor, or as the completion signal of a distributed termination protocol. Clocks are a general and flexible means for bundling asynchronous behaviour into intervals. Secondly, the concept of time underlying the use of clocks is *qualitative* since it is not the absolute occurrence time or duration of actions that is constrained but their relative ordering and sequencing wrt. clocks. Our approach postpones the analysis of timing inequations until a concrete realization is fixed assigning specific delays to actions and clock distances. For timing-critical aspects, of course, a particular delay constraint would be imposed on a particular clock already at specification time. Finally, clocks admit a *local* notion of time since in different subprocesses independent clocks can be used, which may or may not be realized referring to the same time base.

The contribution of this paper is to introduce the syntax and semantics of the process algebra *PMC*, which is a non-trivial extension of CCS [Mil89a] by multiple clocks. The semantics of *PMC* is based on transition systems with separate action and clock transitions. Actions are *insistent*, so that local constraints on the progress of clocks can be expressed. The important features of *PMC* demonstrated in this paper are its flexibility in expressing timing constraints, and the fact that it admits a complete axiomatization of strong bisimulation equivalence for a class of finite-state processes and of observation congruence for finite processes.

2 Syntax and Semantics of PMC

In PMC concurrent systems are described by their ability to perform *actions* and synchronize with *clocks*. As in CCS we assume a set of *action names* \mathcal{A} and their complemented versions $\bar{\mathcal{A}}$. Let \dashv be a bijection between \mathcal{A} and $\bar{\mathcal{A}}$ whose inverse is also denoted by \dashv . We assume an additional *silent* action τ , which is not in $\mathcal{A} \cup \bar{\mathcal{A}}$ and take the set of *actions* to be $\text{Act} =_{\text{def}} \mathcal{A} \cup \bar{\mathcal{A}} \cup \{\tau\}$. Communication takes place as a synchronization between complementary actions a and \bar{a} ; the actions $a \in \mathcal{A}$ are considered to be *input actions* and the actions \bar{a} to be *output actions*. In the sequel we let α, β, \dots range over Act .

In addition to the ordinary actions of CCS, PMC assumes a finite set of clocks Σ the elements of which are ranged over by $\rho, \sigma, \sigma', \sigma_1$ etc. Whereas the actions are used for two-by-two synchronization between parallel processes, the clocks enforce broadcast synchronization in which all processes of a parallel composition must take part. In fact, clocks mimic the properties of time in that the effect of a clock tick reaches through almost all syntactic operators.

Let x range over a set of *process variables*. Process terms t are generated from the following grammar:

$$t ::= \mathbf{0} \mid \alpha.t \mid t_0 + t_1 \mid t \setminus a \mid [t_0]\sigma(t_1) \mid t \uparrow \sigma \mid x \mid \text{rec } x.t$$

Roughly, the meaning of the process operators, in terms of their ability to perform actions or to take part in clock ticks, is as follows. *Nil*: $\mathbf{0}$ is the process which can do nothing, neither an action nor does it admit a clock tick. *Insistent prefix*: $\alpha.t$ is the process which performs α and then behaves as t ; it prevents all clocks from ticking, which motivates calling it ‘insistent’ prefix. The term ‘insistent’ is taken from Hennessy [Hen93]. Prefixes that stop time from progressing also have been called *urgent* [BL91] or *immediate* [NS91a]. *Sum*: $t_0 + t_1$ is the process which must behave as any of t_0 or t_1 , the choice being made with the first action. *Composition*: $t_0 \mid t_1$ represents t_0 and t_1 performing concurrently with possible communication. *Restriction*: $t \setminus a$ behaves like t but with actions a, \bar{a} not allowed to occur. Each one of the processes $t_0 + t_1$, $t_0 \mid t_1$, and $t \setminus a$ takes part in a clock tick σ by having all of its components t_0, t_1, t take part in it. *Timeout*: $[t_0]\sigma(t_1)$ is the process which behaves like t_0 if an initial action of t_0 is performed or a clock tick different from σ occurs; if however σ occurs first it is transformed into t_1 , whence the name ‘timeout’. *Ignore*: The process $t \uparrow \sigma$ behaves just like t but it will always take part in a σ clock tick without changing its state. *Recursion*: $\text{rec } x.t$ is a distinguished solution of the process equation $x = t$.

The notion of a *closed* term and the set of *free variables* of a term are defined as usual. A variable x is *weakly guarded* in a term t if each occurrence of x is within a subexpression t' of t which is in the scope of a $\alpha.t'$ or of a $[\cdot]\sigma(t')$. If we require $\alpha \neq \tau$ in this definition, then x is *strongly guarded* in t . A term t is weakly/strongly guarded if every variable occurring in t is weakly/strongly guarded. We will use the symbol \equiv to denote syntactic equality.

The semantics of PMC is given by a *labelled transition system* $\mathcal{T} = (\mathcal{P}, \mathcal{L}, \rightarrow)$, where \mathcal{P} is the set of closed process terms, $\mathcal{L} =_{\text{def}} \text{Act} \cup \Sigma$ is the set of labels, and $\rightarrow \subseteq \mathcal{P} \times \mathcal{L} \times \mathcal{P}$ is the transition relation. In distinguishing between pure action

$\alpha.p \xrightarrow{\alpha} p$	$\frac{p \xrightarrow{\alpha} p'}{p + q \xrightarrow{\alpha} p'}$	$\frac{q \xrightarrow{\alpha} q'}{p + q \xrightarrow{\alpha} q'}$
$\frac{p \xrightarrow{\alpha} p' \quad q \xrightarrow{\bar{\alpha}} q'}{p \mid q \xrightarrow{\bar{\alpha}} p' \mid q'}$	$\frac{p \xrightarrow{\alpha} p'}{p \mid q \xrightarrow{\alpha} p' \mid q}$	$\frac{q \xrightarrow{\alpha} q'}{p \mid q \xrightarrow{\alpha} p \mid q'}$
$\frac{p \xrightarrow{\alpha} p'}{p \setminus a \xrightarrow{\alpha} p' \setminus a} (\alpha \neq a, \alpha \neq \bar{a})$		$\frac{t[rec\ x.t/x] \xrightarrow{\alpha} p}{rec\ x.t \xrightarrow{\alpha} p}$
$\frac{p \xrightarrow{\alpha} p'}{[p]\sigma(q) \xrightarrow{\alpha} p'}$	$\frac{p \xrightarrow{\alpha} p'}{p \uparrow \sigma \xrightarrow{\alpha} p' \uparrow \sigma}$	

Fig. 1. Action rules.

and pure clock/time transitions the semantics follows the popular pattern for timed process algebras [NS91b]. Of course, there are other ways of incorporating time. We mention ICPA [AM93a] where the transitions carry both action and time information. The transition relation \rightarrow of PMC is defined in Plotkin style as the least relation closed under the set of *action rules* and *clock progress rules* given in Figs. 1 and 2.

$\frac{p \xrightarrow{\sigma} p' \quad q \xrightarrow{\sigma} q'}{p + q \xrightarrow{\sigma} p' + q'}$	$\frac{p \xrightarrow{\sigma} p' \quad q \xrightarrow{\sigma} q'}{p \mid q \xrightarrow{\sigma} p' \mid q'}$	$\frac{p \xrightarrow{\sigma} p'}{p \setminus a \xrightarrow{\sigma} p' \setminus a}$	$\frac{t[rec\ x.t/x] \xrightarrow{\sigma} p}{rec\ x.t \xrightarrow{\sigma} p}$
$[p]\sigma(q) \xrightarrow{\sigma} q$	$\frac{p \xrightarrow{\sigma'} p'}{[p]\sigma(q) \xrightarrow{\sigma'} p'} (\sigma \neq \sigma')$		
$p \uparrow \sigma \xrightarrow{\sigma} p \uparrow \sigma$		$\frac{p \xrightarrow{\sigma'} p'}{p \uparrow \sigma \xrightarrow{\sigma'} p' \uparrow \sigma} (\sigma \neq \sigma')$	

Fig. 2. Clock progress rules.

The action rules (Fig. 1) for the processes $\alpha.p, p + q, p \mid q, p \setminus a$, and $rec\ x.t$ follow the usual rules for CCS. Also, the clock progress rules (Fig. 2) for these processes require little comment. The idea is that a clock tick is a global time synchronization which synchronously affects all subterms of a process. The action and clock progress rules for $p \uparrow \sigma$ and $[p]\sigma(q)$ reflect the informal explanation given above.

It will be convenient to extend the timeout operator to (possibly empty) sequences $\underline{\sigma} = \sigma_1 \cdots \sigma_n$ of clocks by the following inductive definition:

$$\begin{aligned} [t] &= t \\ [t]\sigma_1(u_1) \cdots \sigma_n(u_n) &= [[t]\sigma_1(u_1) \cdots \sigma_{n-1}(u_{n-1})]\sigma_n(u_n). \end{aligned}$$

We shall sometimes use the vector notation $[t]\underline{\sigma}(u)$ to abbreviate $[t]\sigma_1(u_1)\cdots\sigma_n(u_n)$.

The following is a list of some important derived constructions which are all special cases of timeout:

$$\begin{aligned} 0_{\underline{\sigma}} &=_{\text{def}} \text{rec } x.[0]\sigma_1(x)\cdots\sigma_n(x), & (\text{relaxed nil}) \\ \alpha :_{\underline{\sigma}} t &=_{\text{def}} \text{rec } x.[\alpha.t]\sigma_1(x)\cdots\sigma_n(x) & (\text{relaxed prefix}) \\ \sigma.t &=_{\text{def}} [0]\sigma(t) & (\text{wait}) \\ \sigma :_{\underline{\sigma}} t &=_{\text{def}} \text{rec } x.[0]\sigma_1(x)\cdots\sigma_n(x)\sigma(t) & (\text{relaxed wait}) \end{aligned}$$

For the clock progress rules of Fig. 2 we notice the absence of a rule for nil and prefix. They both stop *all* clocks; 0 has the rather dramatic effect of stopping all clocks indefinitely, it is a time-lock. In contrast, the process $0_{\underline{\sigma}}$ in the above table is a relaxed version of nil which does not perform any action but allows the clocks in $\underline{\sigma}$ to proceed. The maximally relaxed nil process, which enables all clocks, is abbreviated by 1 , i.e. $1 =_{\text{def}} 0_{\sigma_1\cdots\sigma_N}$, where $\Sigma = \{\sigma_1, \dots, \sigma_N\}$. If Σ is empty, then 1 and 0 coincide. The process $\alpha :_{\underline{\sigma}} t$ above is a *relaxed* version of a prefix: it lets clocks $\underline{\sigma}$ tick away freely without changing its state, until an α action occurs which transforms it into t . The derived process $\sigma.t$ may be the most common construct in applications: it waits for the clock σ before proceeding with t , and in doing so it is stopping all other clocks. Finally, $\sigma :_{\underline{\sigma}} t$ is a *relaxed* wait: it waits for clock σ but allows all clocks in $\underline{\sigma}$ to tick.

It should be mentioned that by a generalization of the above constructions for ‘relaxing’ processes the ignore $p \uparrow \sigma$ actually can be eliminated from *closed* processes by induction on the structure of p . This does not mean that \uparrow is redundant syntactically, since it is not a derived operator and the elimination does not work on open terms.

3 Example: Signal Analyzer

The initial ideas and motivation leading to PMC developed from an attempt formally to specify the Brüel & Kjær 2145 Vehicle Signal Analyzer.¹

The Brüel & Kjær 2145 Vehicle Signal Analyzer is an instrument for measuring noise from machines with rotating objects such as cars and turbines. The instrument receives input from microphones and tachometers, computes running spectra of amplitudes of frequencies, and presents the results on a screen. Various measuring scenarios and presentation modes are available. The analyzer is operated in real-time and the results are visualized on the screen as they are computed.

In this example, to illustrate an application of PMC, we will simplify the real design and describe only that part of the instrument that collects sound samples from one microphone, speed pulses from one tachometer, and computes a speed-related spectrum. Figure 3 gives an overview of the simplified system. It features seven communication channels represented by the solid lines, and three clocks represented by dashed lines. The signal analyzer reads in sound samples along channel s , tacho pulses along p , and outputs on channel $srsp$ the speed-related spectrum computed

¹ This case study is done in co-operation with the manufacturing company and is part of the Co-design Project at the Department of Computer Science, DTH.

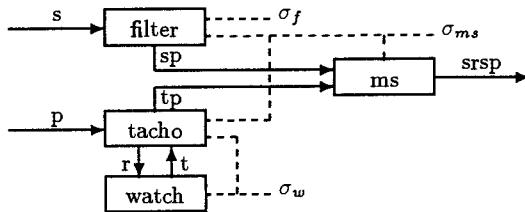


Fig. 3. The Signal Analyzer.

from the samples and the tacho information. The clocks reflect central timing aspects of the system behaviour. The sampling rate is modelled by the clock σ_f which ticks with a fixed frequency determined by the input bandwidth. Another clock σ_w is used for measuring the time between tacho pulses. It also ticks with a fixed frequency determining the precision by which time is measured in the instrument. A third clock σ_{ms} is used for synchronizing the exchange of information between the three processes 'filter', 'tacho' and 'ms'.

Abstracting away from the actual values sent along the channels and concentrating on the communication patterns and the real-time aspects, the system consists of the following four processes:

- The process 'filter' reads after each tick of σ_f a sample on s , computes a spectrum (modelled by a τ) and then either delivers the spectrum on sp in response to a tick of σ_{ms} or waits for the next sample to arrive.
- The process 'watch' keeps track of the time based on ticks of the clock σ_w . It can be reset using channel r and the current time can be read on channel t .
- The process 'tacho' records the number of tacho pulses arriving on p and, whenever the clock σ_{ms} ticks, delivers the pulse count on tp together with the time distance since the last delivery (as measured by 'watch'). For safe real-time operation 'tacho' must occasionally prevent σ_w from ticking.
- The measuring process 'ms' collects, with every tick of the clock σ_{ms} , a spectrum on sp , a tacho count and time distance on tp . From these it computes a speed-related spectrum finally delivered on $srsp$. Depending on the spectrum received and the value of the tacho count, this can be more or less involved. We model this by the 'ms' process making an internal choice between delivering the result immediately or performing some lengthy internal computation (modelled by a sequence of τ 's below) before doing so.

The following is a PMC description of the system:

$$\begin{aligned}
 \text{filter} &= [\sigma_f . s . \text{filter}] \sigma_{ms}(\overline{sp} . \text{filter}) \\
 \text{watch} &= [r . \text{watch} + \bar{t} . \text{watch}] \sigma_w(\text{watch}) \\
 \text{tacho} &= [p :_{\sigma_w} \text{tacho}] \sigma_{ms}(t :_{\sigma_w} \bar{r} . \overline{tp} :_{\sigma_w} \text{tacho}) \\
 \text{ms} &= \sigma_{ms} . sp . tp . (\tau . \overline{srsp} . \text{ms} + \tau . \cdots \tau . \overline{srsp} . \text{ms})
 \end{aligned}$$

The filter, tacho, and watch processes constitute the input system

$$\text{INP} = (\text{filter} \uparrow \sigma_w) | (\text{tacho} \uparrow \sigma_f | \text{watch} \uparrow \sigma_f \uparrow \sigma_{ms}) \setminus t \setminus r$$

and the complete system is

$$\text{SYS} = (\text{INP} \mid \text{ms} \uparrow \sigma_w \uparrow \sigma_f) \setminus \text{sp} \setminus \text{tp}.$$

With the formal description of the system at hand we can make precise the rôle of the three clocks $\sigma_w, \sigma_{ms}, \sigma_f$.

For correct real-time operation it is important to make sure that with every input action t ‘tacho’ obtains the exact number of σ_w ticks arrived since the last time it read the watch. This implies that *no* tick of σ_w must fall between reading the current time of the watch with action t and resetting it with action \bar{r} . This real-time requirement is conveniently dealt with in PMC by using an insistent prefix after \bar{r} in the tacho process, which prevents σ_w from ticking between reading and resetting of the watch. Using ordinary actions instead of a clock to distribute time signals, this mutual exclusion between the tacho and the watch process would have to be encoded using a protocol.

The clock σ_{ms} ensures that whenever ‘filter’, ‘tacho’ and ‘ms’ are ready, consistent pairs of spectra and tacho values are sent. This relies on the broadcast feature of clocks forcing all parties to synchronize. Using normal actions an effect like this requires a rather complex protocol, and it is quite hard to ensure that the values from ‘filter’ and ‘tacho’ never arrive out of synchrony.

Finally, the clock σ_f also plays some rôle. It might be argued that as long as the samples from the environment arrive at the right speed on the channel s , the clock σ_f is unnecessary. However, if the samples are available from the environment as the values of a state variable that can always be read, it is important that this happens only at certain well-defined points. This is enforced by the clock. Moreover, one could imagine adding another filter process sampling a parallel input channel in synchrony with the first one. The synchronization of both filters, which comes for free with the clock σ_f , otherwise would have to be encoded via a protocol.

4 Example: Synchronous Hardware

In hardware design one is dealing frequently with architectures consisting of a number of interconnected synchronous systems that are all driven by independent, i.e. local clocks. Such systems are called *multi-clock synchronous systems* or *globally-asynchronous, locally-synchronous machines* [Cha87]. The synchronous subsystems exchange data via communication buffers which decouple the computations and compensate for different relative clock speeds. The simplest case of a communication buffer is the *input synchronizer* as shown in Fig. 4.

The input synchronizer S is prepared to accept input data on x from the environment at any time, and offers it to the synchronous system at its output y only on the next tick of the local clock σ_2 . Of any sequence of input data arriving before the clock tick only the most recent input is transmitted. Since the output value the synchronizer offers on y may change with every clock tick the output behaviour will not be preserved by clock transitions. It can be shown [AM93b] that this cannot be expressed merely with wait, nil, or prefixes, be they relaxed or not. However, with the

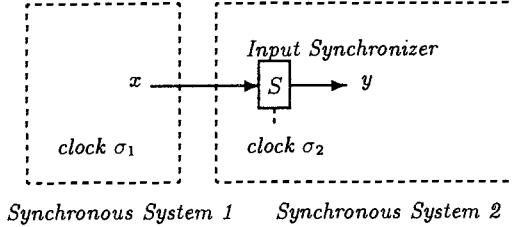


Fig. 4. An input synchronizer.

general timeout operator the synchronizer can be defined:

$$S(v, w) =_{\text{def}} [x?d . S(d, w) + y!w . S(v, w)] \sigma_2(S(v, v)),$$

where we encode value-passing as in CCS by the abbreviations

$$x?d . p \equiv_{\text{def}} \sum_{d \in D} x_d . p \quad y!w . p \equiv_{\text{def}} \bar{y}_w . p,$$

assuming that D is a finite domain of values and x, y families of distinct actions indexed by the elements $d, w \in D$. The synchronizer process $S(v, w)$ has two parameters: the first one, v , represents the state of the input line which can be changed to a new value d by an $x?d$ action at any time. The second parameter w is the state of the output line. It is passed on to the synchronous system with $y!w$ at any time, and is updated with the value of the first parameter whenever clock σ_2 ticks.

The input synchronizer can be used to connect up two single-clock synchronous systems, say SC_1 and SC_2 , running with independent clocks σ_1, σ_2 :

$$MC(v) =_{\text{def}} SC_1 \uparrow \sigma_2 \mid (S(v, v) \mid SC_2) \uparrow \sigma_1.$$

How single-clock synchronous systems can be modelled in PMC is explained in [AM93b]. The idea is to use the insistent prefixes to synchronize the function blocks globally and force the timing constraint upon the clock signal. This leads to a compositional calculus of synchronous systems, in which all the subcomponents of SC_i themselves are special cases of synchronous systems.

5 Axiomatization

Having set up the syntax and semantics of PMC, we are now going to present a formal calculus for reasoning about equivalence of PMC processes. We wish to axiomatize two notions of equivalence, viz. strong bisimilarity and observation congruence. These notions carry over naturally from CCS [Mil89a] by treating clock and ordinary action transitions in the same way:

Definition 1. A relation $S \subseteq \mathcal{P} \times \mathcal{P}$ is a *strong bisimulation* if it is symmetric, and for all $(p, q) \in S$ and $l \in \mathcal{L} = \text{Act} \cup \Sigma$, whenever $p \xrightarrow{l} p'$ then for some q' , $q \xrightarrow{l} q'$ and $(p', q') \in S$. Two processes p and q are *strongly bisimilar* if $p \sim q$, where \sim is the union of all strong bisimulations.

Example 1. The following processes are equivalent formulations of filter and watch from Sec. 3:

$$\begin{aligned}\text{filter}' &\equiv \text{rec } x. \sigma_f :_{\sigma_{ms}} s . \tau . x + \sigma_{ms} :_{\sigma_f} \overline{sp} . x \\ \text{watch}' &\equiv (\text{rec } x. r . x + \bar{t} . x) \uparrow \sigma_w,\end{aligned}$$

i.e. we have $\text{filter} \sim \text{filter}'$ and $\text{watch} \sim \text{watch}'$. Notice that the latter equivalence holds only because we are abstracting away from values. If values were considered our watch surely would not ignore time.

To define observation congruence we need a few auxiliary concepts: If $s \in \mathcal{L}^* = (Act \cup \Sigma)^*$ then $\hat{s} \in (\mathcal{L} \setminus \tau)^*$ is the sequence obtained from s by deleting all occurrences of τ . For $s = s_1 \dots s_n \in \mathcal{L}^*$ ($n \geq 0$), we write $p \xrightarrow{s} q$ if $p \xrightarrow{s_1} \dots \xrightarrow{s_n} q$, and $p \xrightarrow{\hat{s}} q$ if $p \xrightarrow{\tau} * \xrightarrow{s_1} \tau * \dots \xrightarrow{s_n} \tau * q$. Note in particular, $p \Rightarrow p$.

Definition 2. A relation $S \subseteq \mathcal{P} \times \mathcal{P}$ is a *weak bisimulation* if it is symmetric, and for all $(p, q) \in S$ and $l \in \mathcal{L}$, whenever $p \xrightarrow{l} p'$ then for some $q' \in S$, $q \xrightarrow{l} q'$ and $(p', q') \in S$. Two processes p and q are *weakly bisimilar* if $p \approx q$, where \approx is the union of all weak bisimulations. *Observation congruence*, written \approx^c , is the largest congruence contained within \approx .

We extend both equivalences to open terms in the usual way by stipulating tEu if for every substitution θ of closed terms for the free variables, $t[\theta]Eu[\theta]$, where E is one of \sim , \approx^c . For processes p, q without clock transitions the definitions coincide with the corresponding notions for CCS, whence no extra equivalences are introduced for the CCS sublanguage of PMC. In other words, PMC is a conservative extension of CCS.

One can show that the offset between \approx and \approx^c is almost of the same nature as for CCS.

Lemma 3. $p \approx^c q$ iff one of the following three equivalent properties holds:

- For all r , $p + r \approx q + r$.
- If $p \xrightarrow{\alpha} p'$ then for some q' , $q \xrightarrow{\alpha} q'$ and whenever $p' \xrightarrow{\alpha} p''$ then $q' \xrightarrow{\alpha} q''$ for some q'' such that $p'' \approx q''$; the same holds symmetrically with p and q interchanged.
- $(p, q) \in S$ where S is any symmetric binary relation on \mathcal{P} with the property that for all $(r, s) \in S$, $\alpha \in Act$, $\sigma \in \Sigma$,
 - (i) if $r \xrightarrow{\alpha} r'$ then for some s' , $s \xrightarrow{\alpha} s'$ and $r' \approx s'$,
 - (ii) if $r \xrightarrow{\alpha} r'$ then for some s' , $s \xrightarrow{\alpha} s'$ and $(r', s') \in S$.

According to the first characterization the reason why \approx fails to be a congruence can be localized in the sum operator. In this respect the situation is precisely as in CCS. The second characterization brings up the difference: while in CCS congruent processes need a strong match for the first τ actions in PMC they need to match any initial clock sequence followed by a τ . The third characterization coincides with one given by Moller and Tofts for observation congruence in TCCS [MT92]. The equivalence between the last two characterizations is due to the following property:

S1	$t + u = u + t$
S2	$t + (u + v) = (t + u) + v$
S3	$t + t = t$
S4	$\alpha \cdot t + 0 = \alpha \cdot t$
S5	$t + 1 = t$
B1	$\lfloor [t] \sigma(u) \rfloor \sigma(v) = \lfloor t \rfloor \sigma(v)$
B2	$\lfloor [t] \sigma(u) \rfloor \sigma'(v) = \lfloor [t] \sigma'(v) \rfloor \sigma(u)$
B3	$\lfloor t \rfloor \sigma(u) + \lfloor v \rfloor \sigma(w) = \lfloor t + v \rfloor \sigma(u + w)$
B4	$\lfloor t \rfloor \sigma(u) + \lfloor 0 \rfloor \underline{\sigma}(v) = t + \lfloor 0 \rfloor \underline{\sigma}(v)$
CR	if $\alpha \cdot u = \alpha \cdot v$ then $\alpha \cdot \lfloor t \rfloor \sigma(u) = \alpha \cdot \lfloor t \rfloor \sigma(v)$
C1	$0 \setminus a = 0$
C2	$(\alpha \cdot t) \setminus a = \begin{cases} 0 & \alpha = a \text{ or } \alpha = \bar{\alpha} \\ \alpha \cdot (t \setminus a) & \text{otherwise} \end{cases}$
C3	$(t + u) \setminus a = t \setminus a + u \setminus a$
C4	$\lfloor t \rfloor \sigma(u) \setminus a = \lfloor t \setminus a \rfloor \sigma(u \setminus a)$
I1	$0 \uparrow \sigma = \lfloor 0 \rfloor \sigma(0 \uparrow \sigma)$
I2	$(\alpha \cdot t) \uparrow \sigma = \lfloor \alpha \cdot (t \uparrow \sigma) \rfloor \sigma((\alpha \cdot t) \uparrow \sigma)$
I3	$(t + u) \uparrow \sigma = t \uparrow \sigma + u \uparrow \sigma$
I4	$\lfloor t \rfloor \sigma'(u) \uparrow \sigma = \lfloor t \uparrow \sigma \rfloor \sigma'(u \uparrow \sigma) \sigma(\lfloor t \rfloor \sigma'(u) \uparrow \sigma)$

Fig. 5. Equational laws for sum, timeout, restriction, and ignore.

$$E \quad t \mid u = \lfloor r \rfloor \sigma_1(t'_1 \mid u'_1) \cdots \sigma_k(t'_k \mid u'_k),$$

where t, u are terms

$$t \equiv \lfloor \sum_I \alpha_i \cdot t_i \rfloor \rho_1(t'_1) \cdots \rho_l(t'_l) \quad u \equiv \lfloor \sum_J \beta_j \cdot u_j \rfloor \sigma_1(u'_1) \cdots \sigma_m(u'_m),$$

such that $\rho_i = \sigma_i$ for $1 \leq i \leq k$, and $\rho_i \neq \sigma_j$ for $k < i$ or $k < j$, and r is the term

$$r \equiv \sum_I \alpha_i \cdot (t_i \mid u) + \sum_J \beta_j \cdot (t \mid u_j) + \sum_{\alpha_i = \bar{\beta}_j} \tau \cdot (t_i \mid u_j).$$

Fig. 6. The expansion law.

R0	$\text{rec } x.t = \text{rec } y.t[y/x]$	y not free in $\text{rec } x.t$
R1	$\text{rec } x.t = t[\text{rec } x.t/x]$	
R2	if $u = t[u/x]$ then $u = \text{rec } x.t$	x guarded* in t
R3	$\text{rec } x.(\lfloor x \rfloor \underline{\sigma}(u) + t) = \text{rec } x.(\lfloor 0 \rfloor \underline{\sigma}(u) + t)$	

* Weakly guarded if $=$ is interpreted as \sim , strongly guarded if it stands for \approx^c . In the latter case t must be regular too.

Fig. 7. Laws for recursion.

T1	$\alpha.\tau.t = \alpha.t$
T2	$\tau.t + t = \tau.t$
T3	$\alpha.(t + [\tau.u]\sigma(v)) + \alpha.u = \alpha.(t + [\tau.u]\sigma(v))$
T4	$\alpha.[\tau.[r]\sigma([\tau.s + t]\rho(u))] + v]_{\kappa}(w) = \alpha.[\tau.[r]\sigma([\tau.s + t]\rho(u))] + v]_{\kappa}(s)_{\kappa}(w)$

Fig. 8. Tau laws.

Proposition 4 Clock Determinism. *If $p \xrightarrow{\sigma} q$ and $p \xrightarrow{\sigma} r$, then $q \equiv r$.*

By definition, \approx^c is a congruence wrt. all operators, including the recursion operator $rec\ x$. It is not difficult to verify that \sim also is a congruence wrt. all operators. Congruicity is important, since it shows that in an axiomatization of both equivalences, \sim , \approx^c , Leibniz' rule of ‘substituting equals for equals’ is sound, that is, if we have proven $t = u$ then t and u can be interchanged in any context. We shall use the symbol \vdash to denote that an equality $\vdash t = u$ is derivable using equational reasoning and the special laws that we shall consider in the sequel.

Theorem 5 Soundness. *The laws of Fig. 5–7 are sound for \sim and \approx^c . The τ laws of Fig. 8 are sound for \approx^c .*

Example 2. Let us prove, by equational reasoning from our axioms, the equivalence in example 1 for the watch process:

$$\begin{aligned}
 \text{watch}' &\equiv A \uparrow \sigma_w \\
 &= (r \cdot A + \bar{t} \cdot A) \uparrow \sigma_w && \text{R1} \\
 &= (r \cdot A) \uparrow \sigma_w + (\bar{t} \cdot A) \uparrow \sigma_w && \text{I3} \\
 &= [r \cdot \text{watch}'] \sigma_w ((r \cdot A) \uparrow \sigma_w) + [\bar{t} \cdot \text{watch}'] \sigma_w ((\bar{t} \cdot A) \uparrow \sigma_w) && \text{I2} \\
 &= [r \cdot \text{watch}' + t \cdot \text{watch}'] \sigma ((r \cdot A) \uparrow \sigma_w + (t \cdot A) \uparrow \sigma_w) && \text{B3} \\
 &= [r \cdot \text{watch}' + \bar{t} \cdot \text{watch}'] \sigma (\text{watch}') && \text{I3, R1.}
 \end{aligned}$$

By rule R2 we conclude $\vdash \text{watch}' = \text{watch}$.

Definition 6. A term t is said to be *regular* if it is built from nil, prefix, sum, time-out, variables and the recursion operator. A process term t is *rs-free* (rs abbreviates ‘recursion through static operators’) if every subterm $rec\ x.u$ of t is regular. A process p is *finite* if it does not contain any recursion or ignore operators.

Theorem 7 Completeness. *If p and q are rs-free processes with $p \sim q$ then $\vdash p = q$ using equational reasoning from the laws of Figs. 5–7 (without CR). If p and q are finite processes and $p \approx^c q$, then $\vdash p = q$ using equational reasoning from the laws of Figs. 5–8.*

The proof of completeness for \sim can be found in [AM93b] and for \approx^c it will appear elsewhere. The proofs are basically an adaptation of Milner’s technique [Mil84, Mil89a] but rather more involved due to a more complicated normal form representation and the larger gap between \approx and \approx^c .

In view of the completeness theorem one might hope that a lot of the mathematical theory of CCS can be carried over easily to PMC. But the situation is not quite so simple. There are some subtle technical complications making PMC a non-trivial extension of CCS.

Firstly, the standard approach extending completeness from finite to finite-state processes [Mil89b] builds on the fact that in CCS unguarded processes can always be transformed into guarded ones. Unfortunately, this property fails to hold for PMC, with the consequence if \approx^c can be completely axiomatized for finite-state processes then a new proof strategy must be found. For instance, take the unguarded process $p \equiv \text{rec } x. [\tau . [\tau . x]\sigma(1)]\sigma(0)$. In every state of p reachable through, possibly zero, τ actions there is a weak σ transition both to 0 and to 1. This property must be enjoyed by any process q weakly bisimilar to p . However, to fulfill this property q must either be infinite state or have a τ loop. To see why, consider a state q_0 such that $q \Rightarrow q_0$ and $q_0 \xrightarrow{\sigma} 0$ which must exist by assumption. But there must also be a state q_1 such that $q_1 \xrightarrow{\sigma} 1$ and q_1 is reachable from q_0 through a sequence of τ 's. This sequence cannot be empty for otherwise $q_1 \equiv q_0$ and we would have $q_0 \xrightarrow{\sigma} 1$ contradicting clock-determinism (Prop. 4). Hence we must have $q_0 \xrightarrow{\tau} q_1$. Now the same argument applies to q_1 , so we could go on constructing a sequence $q_0 \xrightarrow{\tau} q_1 \xrightarrow{\tau} q_2 \xrightarrow{\tau} \dots$. But this means that q either has a τ loop or is infinite state.

Secondly, the sum $+$ which is a dynamic operator in CCS, has *static* behaviour wrt. time steps, i.e. it does not disappear after any number of clock transitions. So, to obtain the transition system of $p + q$ from the transition systems of p and q we take the *disjoint union* with respect to ordinary action transitions, but the *product* with respect to clock transitions. This means that in the equational characterization of sums $p + q$ more equations must be added than are needed for the pure CCS fragment. In connection with recursion the situation becomes even more involved: Because of the static nature of $+$ wrt. to clock transitions there are regular processes with infinite syntactic unfolding. For instance the process $P \equiv \text{rec } x. [0]\sigma(x + (p \uparrow \sigma))$ admits the infinite transition sequence

$$P \xrightarrow{\sigma} P + p \uparrow \sigma \xrightarrow{\sigma} \dots \xrightarrow{\sigma} P + p \uparrow \sigma + \dots + p \uparrow \sigma \xrightarrow{\sigma} \dots$$

producing bigger and bigger terms. In the pure CCS fragment, on the other hand, a regular process always has a finite syntactic unfolding. Nevertheless regular PMC processes have a finite number of states modulo \sim (cf. [AM93b]).

6 Related Work

We begin with a remark on terminology. The most distinguished feature of PMC is the notion of ‘clock’. For most purposes this term would denote a means for measuring time in order to time-stamp observations. In the process language CIPA [AM93a] or in the timed automata of Alur and Dill [AD91] clocks are used in this sense. In PMC, however, the intended interpretation of ‘clock’ is more like that of a hardware clock, viz. a global signal used to synchronize asynchronous computations in a lock-step fashion.

We believe that in the context of asynchronous process calculi the concept of multiple synchronization clocks — in our sense — is novel. Yet, it is not entirely new as it has been used already in synchronous real-time description languages. LUSTRE [HPOG89] is a language for synchronous data-flow with multiple clocks, where all clocks are derived from a master clock through boolean expressions. LUSTRE was developed originally for real-time programming but is used also for describing digital circuits. Another quite successful real-time language with a multi-form notion of time is ESTEREL [BC84]. It must be noted however, that in both these languages clocks are not built-in; they are ordinary signals or variables, not an independent semantical concept as in PMC. A synchronous language where clocks do possess genuine semantical meaning with an associated ‘clock calculus’ is SIGNAL [BBG93].

The obvious — albeit not stringent — path towards a technical comparison with other published work is to view PMC with a single clock as a discrete time process algebra, where a time delay of size n corresponds to n successive clock ticks. For instance, if we fix a particular clock σ we can define timing operators $(n).t$ and $\delta.t$ as follows

$$\begin{aligned}(0).t &\equiv t \\ (n+1).t &\equiv \sigma . (n).t \\ \delta.t &\equiv \text{rec } x. \lfloor t \rfloor \sigma(x)\end{aligned}$$

with n ranging over natural numbers. For every n , the process $(n).t$ waits n clock transitions of σ before it evolves into t , and until then it remains quiet. The process $\delta.t$ is a delayed version of t which allows σ to proceed until such time as the environment is ready to communicate with it. These constructs are taken as primitives in the timed process calculus TCCS of Moller and Tofts [MT90]. In [MT92] a complete axiomatization for TCCS of observation congruence on finite, sequential processes is presented. Both PMC and TCCS use insistent action prefixes, but where PMC has a timeout operator to produce relaxed actions, relaxed behaviour is introduced in TCCS by a different nonstandard primitive, the *weak* sum $p \oplus q$. It behaves exactly as $p + q$ for ordinary actions, but in contrast to $+$ forces both components to take part in a time transition only if both can do a time transition together. If one of p and q does not admit a time transition it is considered *stopped*, in which case the other process can engage in a time step all by itself while the stopped process is simply dropped from the computation. As hinted at in [MT90] the \oplus plays an important role in obtaining an expansion law for TCCS, which in turn is crucial for proving completeness. It is interesting to note that the equational axiomatization of PMC seems to be considerably simpler than that of TCCS where the expansion law for parallel composition has to consider various special cases (to do with \oplus) while in PMC one single equation scheme suffices.

It is possible to view discrete time TCCS as a subcalculus of PMC modulo the following syntactic encoding of \oplus :

$$(s \oplus t)^* \equiv_{\text{def}} \begin{cases} s^* + \lfloor t^* \rfloor \sigma(1) & \text{if } s^* \xrightarrow{\sigma} \text{ and } t^* \not\xrightarrow{\sigma} \\ \lfloor s^* \rfloor \sigma(1) + t^* & \text{if } s^* \not\xrightarrow{\sigma} \text{ and } t^* \xrightarrow{\sigma} \\ s^* + t^* & \text{otherwise.} \end{cases}$$

The TCCS constructs $(n).t$ and $\delta.t$ are replaced by the definitions given above. All other constructs are represented in PMC by their respective equivalents. It can be shown that the condition $\not\rightarrow$ in the encoding of \oplus can be decided on the syntactic structure, so that $(\cdot)^*$ is in fact a well-defined syntactic operation [AM93b] on closed terms. We conjecture that for closed TCCS processes p with all variables guarded the operational behaviour of the encoding p^* in PMC is precisely the one obtained for p in TCCS.

A rather different class of timed process algebras is that with *relaxed* actions and *maximal progress* [Wan90, HR91]. These principles reflect a rigid two-phase view of real-time execution: In the first phase a component is allowed to perform an arbitrary but finite number of internal communications at zero time cost. When all internal chatter has ceased, i.e. the component has stabilized, time is allowed to proceed in the second phase. The duration of the second phase is the amount of time elapsing until the component again becomes internally unstable. In [NS91b] this two-phase model is generalized to an arbitrary set of *urgent* actions for which maximal progress is enforced. These urgent actions play the role of internal communication in that these actions must be performed before time is allowed to proceed.

This two-phase model cements a *globally-synchronous, locally-asynchronous* type of behaviour. The major mode is synchronous operation since the phases of asynchronous cooperation are bundled together when all subcomponents of a process synchronize to let time advance. In PMC we adopt a more flexible scheme which allows us to localize the notion of time and progress. We can thus obtain simpler and more abstract specifications for distributed systems covering not only globally synchronous, locally asynchronous systems but also the class of *globally-asynchronous, locally-synchronous* behaviour (cf. Sec. 4). It might be possible to extend the maximal progress approach in this direction, an idea suggested in [Hen93], but not without major modification such as ‘localizing’ maximal progress in some appropriate way.

Some remarks on the timeout operators are in order. Our timeouts are an extension to multiple clocks of Nicollin and Sifakis’ timeout introduced originally with the process algebra ATP. In [NS90] they present a complete axiomatization for ATP of strong bisimulation equivalence. ATP is rather like a single clock version of PMC but there are some notable differences. ATP is restricted to rs-free guarded processes without time-locks but has a generalized restriction and parallel composition. We mention that although the syntax of PMC is more involved due to multiple clock constructs the normal form representation seems to be more uniform than in ATP. In PMC only one normal form scheme needs to be handled while in ATP three different cases are treated separately.

There are other “time-insistent” variants of timeouts used in the literature which differ from our’s basically in that for $[p]\sigma(q)$ to perform a σ time step the process p must not prevent time from progressing, which is not the case here. Examples are the constructs $p \xrightarrow{d} q$ of Nicollin and Sifakis [NS91a] and $[p](q)$ of Hennessy and Regan [HR91]. The relaxed time behaviour of $[p]\sigma(q)$ wrt. p is important for PMC as it allows us to derive from it a number of useful “time-relaxed” constructs such as relaxed prefixes. With a time-insistent timeout these relaxed prefixes would have to be added

to the language as primitives, resulting in additional axioms and more complicated normal forms. For instance, in [NS91a] the normal form has to treat separately three different cases. Another central design decision simplifying the normal form of PMC processes is that the clock transitions of $[p]\sigma(q)$ for clocks $\sigma' \neq \sigma$ are treated in the same way as ordinary actions of p , i.e. they remove the timeout.

7 Conclusion

We have presented an extension of CCS by multiple clocks, timeout, and ignore operators, and we have given a complete axiomatization of strong bisimulation equivalence for a class of finite-state processes and of observation congruence for finite processes. The process algebra PMC was developed to capture the quantitative nature of real-time constraints and it is aimed at applications in which real-time requirements are few but essential. We believe that PMC offers a promising compromise between expressiveness and realizability or executability. PMC is currently being used as a specification language in an industrial case study at the Department of Computer Science, DTH. A prototype implementation of a value-passing version of PMC is under development, using the ML-Kit [BRTT93].

Acknowledgement

The authors would like to thank Anders P. Ravn, Matthew Hennessy, Gerard Berry, and Faron Moller for various comments, Anders in particular for his encouragement. Thanks are also due to the referees for their criticism and various suggestions for improving the paper. Both authors have been supported by The Danish Technical Research Council, the second author also by the European Human Capital and Mobility Program, network *EuroFORM*.

References

- [AD91] R. Alur and D. Dill. The theory of timed automata. In de Bakker et al. [dBHdRR91], pages 45–73.
- [AM93a] L. Aceto and D. Murphy. On the ill-timed but well-caused. In E. Best, editor, *Proc. Concur'93*, pages 97–111. Springer LNCS 715, 1993.
- [AM93b] H. R. Andersen and M. Mendler. A process algebra with multiple clocks. Technical Report ID-TR:1993-122, Department of Computer Science, Technical University of Denmark, August 1993.
- [BBG93] A. Benveniste, M. Le Borgne, and P. Le Guernic. Hybrid systems: The SIGNAL approach. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems*, pages 230–254. Springer LNCS 736, 1993.
- [BC84] G. Berry and L. Cosserat. The ESTEREL synchronous programming language and its mathematical semantics. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, pages 389–448. Springer LNCS 197, 1984.

- [BK90] J.C.M. Baeten and J.W. Klop, editors. *Proceedings of CONCUR '90*, volume 458 of *LNCS*. Springer-Verlag, 1990.
- [BL91] T. Bolognesi and F. Lucidi. Timed process algebras with urgent interactions and a unique powerful binary operator. In de Bakker et al. [dBHdRR91], pages 124–148.
- [BRTT93] L. Birkedal, N. Rothwell, M. Tofte, and D. N. Turner. The ML Kit, Version 1. Technical Report, DIKU, March 1993.
- [Cha87] Daniel M. Chapiro. Reliable high-speed arbitration and synchronization. *IEEE Transaction on Computers*, C-36(10):1251–1255, October 1987.
- [dBHdRR91] J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors. *Real-Time: Theory in Practice*, volume 600 of *LNCS*. Springer-Verlag, 1991.
- [Hen93] M. Hennessy. Timed process algebras: A tutorial. Technical Report 93:02, Department of Computer Science, University of Sussex, January 1993.
- [HPOG89] N. Halbwachs, D. Pilaud, F. Ouabdesselam, and A.-C. Glory. Specifying, programming and verifying real-time systems using a synchronous declarative language. In *Workshop on automatic verification methods for finite state systems*, Grenoble, France, June 12–14 1989. Springer LNCS 407.
- [HR91] M. Hennessy and T. Regan. A process algebra for timed systems. Computer Science Technical Report 91:05, Department of Computer Science, University of Sussex, April 1991. To appear in *Information and Computation*.
- [Klu91] A. S. Klusener. Abstraction in real time process algebra. In de Bakker et al. [dBHdRR91], pages 325–352.
- [Lia91] Chen Liang. An interleaving model for real-time systems. Technical Report ECS-LFCS-91-184, Laboratory for Foundations of Computer Science, University of Edinburgh, November 1991.
- [Mil84] Robin Milner. A complete inference system for a class of regular behaviours. *J. of Computer and System Sciences*, 28(3):439–466, June 1984.
- [Mil89a] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.
- [Mil89b] Robin Milner. A complete axiomatisation for observational congruence of finite-state behaviours. *Information and Computation*, 81:227–247, 1989.
- [MT90] Faron Moller and Chris Tofts. A temporal calculus of communicating systems. In Baeten and Klop [BK90], pages 401–415.
- [MT92] F. Moller and Ch. Tofts. Behavioural abstraction in TCCS. In W. Kuich, editor, *Proc. ICALP'92*, pages 559–570. Springer LNCS 623, 1992.
- [NS90] X. Nicollin and J. Sifakis. The algebra of timed processes ATP: theory and application. Technical Report RT-C26, LGI-IMAG, Grenoble, France, December 1990.
- [NS91a] X. Nicollin and J. Sifakis. From ATP to timed graphs and hybrid systems. In de Bakker et al. [dBHdRR91], pages 549–572.
- [NS91b] X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In de Bakker et al. [dBHdRR91], pages 526–548.
- [SDJ⁺91] S. Schneider, J. Davies, D.M. Jackson, G.M. Reed, J.N. Reed, and A.W. Roscoe. Timed CSP: Theory and practice. In de Bakker et al. [dBHdRR91], pages 526–548.
- [Wan90] Yi Wang. Real-time behaviour of asynchronous agents. In Baeten and Klop [BK90].

Foundational Issues in Implementing Constraint Logic Programming Systems

James H. Andrews

Dept. of Computing Science

Simon Fraser University

Burnaby, BC, Canada V5A 1S6

`jamie@cs.sfu.ca`

Abstract: Implementations of Constraint Logic Programming (CLP) systems are often incomplete with respect to the theories they are intended to implement. This paper studies two issues that arise in dealing with these incomplete implementations. First, the notion of incomplete “satisfiability function” (the analogue of unification) is formally defined, and the question of which such functions are reasonable is studied. Second, techniques are given for formally (proof-theoretically) specifying an intended CLP theory or a characterizing an existing CLP system, for the purpose of proving soundness and completeness results. Notions from linear logic and the notion of Henkiness of the theory are shown to be important here.

1 Introduction

The semantics of Constraint Logic Programming (CLP) languages is now well-understood. Implementations of CLP languages, however, are often not complete with respect to their intended semantics. In this paper, I study the theory of such incomplete implementations, by giving a recursion-theoretic, rather than model-theoretic, basis for CLP operational semantics. Based on this work, I then study techniques for specifying the *intended* theory of a CLP language, or giving characterizations of the *actual* theory implemented by a CLP language.

A simple example shows that some CLP implementations are necessarily incomplete. Consider a first order language, structure and theory in which the terms encode lambda-expressions, and in which equality $=$ holds between two terms iff they are $\beta\eta$ -equivalent. The theory can be made satisfaction-complete, and the structure is solution-compact. The CLP scheme [JL87] defines a theoretical operational semantics for this theory such that a ground query $s = t$ fails iff s and t are not $\beta\eta$ -equivalent. However, in practice we have no complete algorithm for testing whether two lambda-terms are $\beta\eta$ -equivalent; so the operational semantics cannot be implemented completely, not even by using the standard breadth-first technique of complete Herbrand-domain logic programming interpreters.

Furthermore, many implementations of CLP languages are incomplete for efficiency reasons; for instance, CLP(R) [JMSY92] implements an efficient but incomplete linear equation solver rather than Tarski’s complex algorithm for deciding real arithmetic.

For these kinds of situations, we need to develop a theoretical framework, incorporating notions of computability, in which we can study issues of how

complete a CLP implementation is. This framework can then be used as a basis for comparing a CLP implementation with its intended theory, or specifying the smaller theory that an incomplete implementation actually implements. This paper is intended to make steps in the direction of such a framework.

In section 2, I present some basic definitions. In section 3, I construct CLP operational semantics based on a class of recursive functions called “satisfiability functions”, which formalize the basic step in CLP languages (the analogue of unification). I also define what it means for a structure to “realize” a satisfiability function, and give a necessary and sufficient, non-logical condition for such a function to be realizable. In section 4, I study various techniques for characterizing CLP theories with proof systems, and show how they could be used to prove the soundness and completeness of implementations. In section 5, I present some conclusions and discuss related work.

2 Basic Definitions

Definition 2.1 A *first order language* \mathcal{L} is a tuple (F, P, V) , where F is a recursive set of function symbols, each with an associated arity, P is a recursive set of predicate names, each with an associated arity, and V is a recursive set of variable names.

Let C be a set of predicate names of \mathcal{L} , $C \subseteq P(\mathcal{L})$. $\text{Constrs}(\mathcal{L}, C)$ is the set of all predicate application formulae from \mathcal{L} formed using a predicate name from C . We also call these predicate application formulae “constraints”.

We define the terms and formulae of \mathcal{L} in the standard logical way; likewise the notions of *structure*, *valuation*, *satisfiability w.r.t. a structure*, *model*, and *theory*.

3 Satisfiability Functions

The basic step in constraint logic programming interpreters (even incomplete ones) is the step which decides whether a new constraint is consistent with the previously-processed constraints. Every CLP interpreter has an algorithm for doing this for its intended theory; if the algorithm returns “true”, the interpreter goes further down the same branch in the search tree, and if it returns “false”, the interpreter backtracks¹. This section studies the theory of such “satisfiability functions”, which will be defined as *partial* recursive functions from finite sets of constraints to results *including* “true” and “false”.

In the first subsection, I define the notion of satisfiability function, and show how an operational semantics can be built on the basis of that notion (rather than the notion of constraint theory) in order to ensure computability. In the

¹This does not necessarily describe completely the operation of all languages referred to as constraint logic programming languages. (For instance, it does not take into account disjunctive constraints.) However, it describes one common framework for CLP operational semantics, and is also the one defined in the standard literature on CLP, e.g. [Mah93].

second subsection, I point out that not all satisfiability functions correspond to actual constraint theories which “realize” them, and thus that the operational semantics arising from them do not define rational logic programming systems. I give, however, a condition on satisfiability functions which is necessary and sufficient for realizability. In the final subsection, I point out that for a given constraint theory, there are either either 0 or 1 maximal satisfiability functions which are realized by it.

3.1 Satisfiability Functions and Operational Semantics

Definition 3.1 A *satisfiability function* (in a language \mathcal{L} with constraint predicates C) is a general recursive function from finite sets of constraints in $\text{Constrs}(\mathcal{L}, C)$ to a set $T \supseteq \{\text{true}, \text{false}\}$.

This definition of satisfiability function is general enough to capture the behaviour of a wide variety of complete and incomplete implementations. We always interpret a result of *false* as “unsatisfiable” and *true* as “satisfiable”; but other results, or no result, is also possible. To capture the very well-behaved satisfiability functions, we make the following definition:

Definition 3.2 A *strict* satisfiability function is one which is never undefined and always returns *true* or *false*.

The satisfiability function associated with basic Prolog, for instance, is strict. Here is a non-strict example.

Example. The basic behaviour of the CLP(R) system [JMSY92] on real-number constraints can be characterized with the following (non-strict) satisfiability function *sat*.

- If S contains a subset T consisting of unsatisfiable, linear constraints, $\text{sat}(S)$ returns *false*.
- Otherwise, if S contains no non-linear constraints, $\text{sat}(S)$ returns *true*.
- Otherwise (i.e. S contains non-linear constraints but its linear constraints are satisfiable), $\text{sat}(S)$ returns *unsure*.

□

In basing an operational semantics on a satisfiability function *sat*, we may want to give a definition that takes into account the various truth values which can act as results of *sat*. We must make the following minimum requirements, following Maher [Mah93].

Definition 3.3 Given a satisfiability function *sat* on (\mathcal{L}, C) whose range is the truth values in T , a binary relation \rightarrow is an *operational transition relation* for *sat* with program P if:

1. It is a relation between *states*, which are either truth values from \mathcal{T} or pairs $\langle G, C \rangle$, where G is a multiset of non-constraint atoms and C is a multiset of constraints;

2. The relation includes the transition

$$\langle G \cup H\theta, C \rangle \rightarrow \langle G \cup B\theta, C \cup C'\theta \rangle$$

if the program P contains (some renaming of) the clause $(H \leftarrow C', B)$, and $\text{sat}(C \cup C'\theta) \simeq \text{true}$; and

3. The relation includes the transition

$$\langle G \cup H, C \rangle \rightarrow \text{false}$$

if for every (renamed) clause in P of the form $(H' \leftarrow C', B)$ and substitution θ such that $H'\theta$ is H , we have that $\text{sat}(C \cup C'\theta) \simeq \text{false}$.

We say that a goal G *succeeds* if $\langle G, \emptyset \rangle \rightarrow^* \langle \emptyset, C \rangle$ and $\text{sat}(C) \simeq \text{true}$; we define *fair* derivations in the usual way and say that G *fails* if every fair derivation ends in the state *false*. Note that if *sat* is strict, then the above definition will yield a unique transition relation, similar to that defined by Maher [Mah93].

Example. Based on the definition of *sat* for CLP(R) above, we can characterize an operational semantics for (an idealized version of a part of) CLP(R) as follows. The transition relation is the unique transition relation for (sat, P) having the additional property that:

- We have the transition

$$\langle G \cup H\theta, C \rangle \rightarrow \langle G \cup B\theta, C \cup C'\theta \rangle$$

if the program P contains (some renaming of) the clause $H \leftarrow C', B$, and $\text{sat}(C \cup C'\theta) \simeq \text{unsure}$.

□

In this operational semantics, even if we are unsure of the satisfiability of the resulting set of constraints (if the system of equations is not linear), we go on as if it were satisfiable. We may wish to say that a goal G is *indeterminate* if it does not fail, but every fair derivation ends in either *false* or *unsure*.

3.2 Realizable Satisfiability Functions

We would like our operational semantics to define sensible logic programming systems, and to achieve that we have to put a condition on the satisfiability functions we use. The condition is “realizability”, and is best defined model-theoretically. In this subsection, we study realizability and give an equivalent condition, “reliability”, which is stated in terms independent of model theory.

Definition 3.4 An \mathcal{L} -structure \mathfrak{N} *realizes* a satisfiability function *sat* if:

1. whenever $\text{sat}(S) \simeq \text{true}$, S is \mathfrak{R} -satisfiable; and
2. whenever $\text{sat}(S) \simeq \text{false}$, S is not \mathfrak{R} -satisfiable.

If \mathfrak{R} realizes sat , we also say that sat implements \mathfrak{R} .

Not every satisfiability function is realizable, not even the strict ones. For instance, if we have a satisfiability function which maps $\{p(x)\}$ onto *false* but $\{p(x), q(x)\}$ onto *true*, then this will not have any realizing structure, because any valuation satisfying $\{p(x), q(x)\}$ will surely satisfy $\{p(x)\}$. An operational semantics based on such a satisfiability function would give unexpected results.

The realizability of sat can be given an equivalent characterization in terms of the theory associated with sat .

Definition 3.5 Θ_{sat} , the *theory associated with sat*, is defined as

$$\{\exists[S] \mid \text{sat}(S) \simeq \text{true}\} \cup \{\neg\exists[S] \mid \text{sat}(S) \simeq \text{false}\}$$

where $\exists[S]$ is the existential closure of the conjunction of the constraints in S .

We have the following simple proposition:

Proposition 3.6 sat is realizable iff Θ_{sat} is consistent.

However, for the purposes of checking a given implementation of a satisfiability function, it would be better to have a more direct method of testing whether it is realizable. The condition called “reliability” allows us to do this. First, some technical definitions.

Definition 3.7 Let S be a set of constraints and let $c \in S$. The *variable sharing class* $S|_c$ is the smallest subset of S such that:

- $c \in S|_c$;
- if $b \in S|_c$, and $a \in S$ shares a free variable with b , then $a \in S|_c$.

Definition 3.8 Let sat be a satisfiability function. A set of constraints S is *sat-covered* if for all $c \in S$, there is a set $T \supseteq S|_c$ such that $\text{sat}(T) \simeq \text{true}$. A set of constraints S is *sat-consistent* if there is some substitution θ such that $S\theta$ is *sat-covered*.

Basically, a set S is *sat-consistent* if, for some θ , every element of this partition of $S\theta$ is satisfiable in any structure realizing sat .

Definition 3.9 A satisfiability function sat is *reliable* if whenever $\text{sat}(S) \simeq \text{false}$, S is not *sat-consistent*.

For non-reliable satisfiability functions, some sets S are considered unsatisfiable despite the fact that some instance of S can be partitioned into sets which are generalizations of sets considered satisfiable. This is a situation which does not meet with our intuitions, and indeed the next theorem proves that reliability is a necessary condition for realizability.

Theorem 3.10 If a satisfiability function sat is realizable, then it is reliable.

Proof (sketch). Assume (toward a contradiction) that sat is realized by some \mathfrak{R} but not reliable. There must be an S and θ such that $sat(S) \simeq \text{false}$ but $S\theta$ is sat -covered. Each variable sharing class T_i in $S\theta$ results in a satisfying valuation v_i ; but the union of these v_i 's is a valuation satisfying $S\theta$, contradicting the assumption that \mathfrak{R} realizes sat .

□

So reliability is necessary for realizability. It is also sufficient, as we will see next.

Theorem 3.11 If a satisfiability function sat is reliable, then it is realizable.

Proof (sketch). By Proposition 3.6, it is sufficient to prove that if sat is reliable, Θ_{sat} is consistent. By compactness, it is therefore sufficient to prove that every finite subset of Θ_{sat} is consistent.

Let \mathbf{S} be a finite subset of Θ_{sat} , where

$$\mathbf{S} = \{\exists[S_1], \dots, \exists[S_j]\} \cup \{\neg\exists[T_1], \dots, \neg\exists[T_k]\}$$

Let \mathfrak{R} be the minimal structure which contains a unique element for every free variable in all the S_i 's, and in which each S_i is satisfiable by a valuation mapping variables to these elements. Clearly this structure is a model of the positive formulae in \mathbf{S} ; we have only to prove that it is a model of the negated formulae too.

Assume, toward a contradiction, that T_i is satisfied by some valuation v in \mathfrak{R} . There is some θ and v' such that $v = \theta v'$, v' maps free variables directly to domain elements, and $T_i\theta$ is satisfied by v' . But by the construction and minimality of \mathfrak{R} , this means that $T_i\theta$ can be partitioned into sets which are free-variable variants of subsets of the S_i 's. T_i is therefore sat -consistent; but we know that $sat(T_i) \simeq \text{false}$, thus contradicting the assumption that sat was reliable.

□

The upshot of this is that if we base operational semantics on satisfiability functions, they are by definition computable; but we are still able to give assurances that they define sensible systems, by proving that the satisfiability function is reliable.

3.3 Maximality Results

Finally, some words about maximality. Given a particular constraint theory, what is the “biggest” satisfiability function which implements it? It turns out that either a theory has a complete, strict satisfiability function, or else there is no maximal satisfiability function which implements it.

We can define maximality by comparing the completeness of two satisfiability functions with respect to a structure.

Definition 3.12 Let sat_1 and sat_2 both implement \mathfrak{R} . We say $sat_1 \sqsubseteq_{\mathfrak{R}} sat_2$, in words “ sat_1 is a better approximation to \mathfrak{R} than sat_2 ”, if:

1. Whenever $sat_1(S) \simeq true$ we have that $sat_2(S) \simeq true$; and
2. Whenever $sat_1(S) \simeq false$ we have that $sat_2(S) \simeq false$.

A satisfiability function sat is a *maximal approximation* to \mathfrak{R} if there is no satisfiability function sat' such that $sat' \neq sat$ and $sat \sqsubseteq_{\mathfrak{R}} sat'$.

Theorem 3.13 A structure \mathfrak{R} has either 0 or 1 maximal approximations.

Proof. If the set S' of satisfiable finite sets of constraints of \mathfrak{R} is recursive, then clearly there is a unique, strict, maximal approximation to \mathfrak{R} . Otherwise, for any sat which implements \mathfrak{R} , we can build a better approximation sat' by allowing more sets to return *true* or *false*.

□

Thus for the example structure given in the Introduction ($\beta\eta$ -equivalence of lambda expressions), there is no maximal implementation. Whenever we have a satisfiability function (and thus a constraint logic programming system) which implements this structure, we can always do better.

4 Specification and Characterization with Proof Systems

We have seen that incomplete implementations of CLP languages are sometimes desirable or even necessary. We have also seen that such incomplete implementations can be given a coherent theoretical basis. But in order to make practical use of incomplete implementations, we need to be able to compare them directly with descriptions of theories.

To do so, we really need *formal* (syntactic) descriptions of the theories to be compared to; the kinds of informal descriptions found in the literature are sometimes too imprecise to be used in formal proofs, and this imprecision is multiplied when we have several groups of interacting constraints (Herbrand, rational tree, integer, real, etc.). There are at least two other practical reasons for developing formal descriptions of CLP languages:

- With increasing prominence of constraint systems, it will become necessary to develop some standard formalism for describing constraint theories, much as BNF was developed to describe programming language syntax.
- Syntactic, or more specifically logical, characterizations of constraint theories will be absolutely necessary to any program-logic system which intends to prove properties of constraint logic programs.

One possible framework for such formal descriptions is proof theory, which has been used to good advantage in the past to describe basic logic programming [HS84, HSH90, MNPS91]. The use of proof theory as a general framework for characterizing CLP, which has not to my knowledge been studied before, is the topic of this section.

There are two ways in which proof-theoretic characterizations could be used:

- (a) to *specify* an intended theory; and
- (b) to *characterize* an existing implementation.

Examples of (a) would include giving an axiomatization of Horn clauses with Presburger arithmetic, to which we could then compare individual CLP implementations. Examples of (b) would include giving a proof-theoretic characterization of the CLP(R) implementation [JMSY92], to see how it looks compared to an axiomatization of real arithmetic.

In this section, I will first discuss some techniques we could use to give proof-theoretic descriptions of CLP theories. One of the major issues that will emerge is the question of whether or not the theory is “Henkin” (has a closed “witness” term for every existential truth). I will then give examples of how such descriptions could be used in proving soundness and completeness of implementations, and in characterizing existing CLP implementations. It will turn out that ideas from linear logic [Gir87] will be relevant to this last point.

4.1 Description Techniques

Here I give two distinct techniques that we might use to describe constraint theories. The first technique is fairly simple but applicable only to Henkin theories; the second is more complex but applicable to both Henkin and non-Henkin theories.

4.1.1 Closed Constraint Technique

In this technique, we characterize the entire theory by characterizing the truth or falsehood of *closed* (ground) constraints. We then rely on proof rules for existential quantification to handle free variables. I will give two examples, then discuss the technique in general.

Example: Herbrand universe logic programming. Proof-theoretic characterizations of basic logic programming have been studied for years. Consider the following example, similar to the system of Miller et al. [MNPS91]. The syntax of goals (G) and definitions (D) is as follows.

$$G ::= t_1 = t_2 \mid p(t_1, \dots, t_n) \mid G \& G \mid G \vee G \mid \exists x G$$

$$D ::= \forall x D \mid p(x_1, \dots, x_n) \leftarrow G$$

Sequents are of the form

$$D_1, \dots, D_m \vdash G$$

and are intended to express “the goal G follows from the definitions D_1, \dots, D_m .” Sequents can be given formal derivations by using the proof rules in Figure 1.

The rules of the proof system both act as a formal and unambiguous description of truth, and give us some intuitive sense of the meaning of the connectives ($G_1 \& G_2$ follows from Γ if both G_1 and G_2 follow from Γ , and so on). We

$$\begin{array}{c}
\frac{\Gamma \vdash G_1 \quad \Gamma \vdash G_2}{\Gamma \vdash G_1 \& G_2} \qquad \frac{\Gamma \vdash G_1}{\Gamma \vdash G_1 \vee G_2} \qquad \frac{\Gamma \vdash G_2}{\Gamma \vdash G_1 \vee G_2} \\
\frac{\Gamma \vdash G[x := t]}{\Gamma \vdash \exists x(G)} \qquad \frac{}{\Gamma \vdash t = t} \\
D, D, \Gamma \vdash G \qquad \frac{D[x := t], \Gamma \vdash G}{\forall x D, \Gamma \vdash G} \qquad \frac{\Gamma \vdash G}{(p(t_1, \dots, t_n) \leftarrow G), \Gamma \vdash p(t_1, \dots, t_n)}
\end{array}$$

Figure 1: A proof-theoretic characterization of a basic logic programming language. Γ is any sequence of definitions.

$$\begin{array}{c}
\frac{\Gamma \vdash a = b}{\Gamma \vdash b = a} \qquad \frac{\Gamma \vdash a = b \quad \Gamma \vdash b = c}{\Gamma \vdash a = c} \qquad \frac{\Gamma \vdash a = b}{\Gamma \vdash s(a) = s(b)} \\
\frac{}{\Gamma \vdash a + 0 = a} \qquad \frac{}{\Gamma \vdash a + s(b) = s(a + b)}
\end{array}$$

Figure 2: Additional rules for a proof-theoretic characterization of a Presburger arithmetic constraint logic programming language.

can prove the soundness and completeness of an SLD-resolution operational semantics with respect to this proof system by two relatively simple structural inductions (e.g. [And89]).

□

Note that we had only to characterize the behaviour of equality formulae, and that we did so by saying that two terms are equal if they are identical. (We must make the assumption that there is at least one closed term in the language, but this is reasonable.)

Example: Presburger arithmetic. Consider the standard language and theory of Presburger arithmetic (i.e., integers constructed from 0 and the successor function $s(\cdot)$, with the arithmetic operation $+$ and the relation $=$). A characterizing proof system can be constructed taking the one for Herbrand logic programming, and adding some more rules to make it a complete proof system for Presburger arithmetic sequents of the form $D_1, \dots, D_m \vdash G$ (see for example [Kle52]). Such additional rules are shown in Figure 2.

□

This specifying proof system does not tell us *how* to solve such constraints or how to construct our implementation; it merely gives a formal and intuitive description of what kinds of constraints we want to solve. It thus acts as a specification of a CLP language for solving Presburger arithmetic constraints. Any implementation, however, would presumably use Presburger's algorithm [Mon76] or some variant of it.

$$\frac{A, \Gamma \vdash G}{\Gamma \vdash G} \qquad \frac{B[x := y], \Gamma \vdash G}{\exists x B, \Gamma \vdash G} \qquad \frac{B, C, \Gamma \vdash G}{B \& C, \Gamma \vdash G} \qquad \frac{x = t, \Gamma \vdash G[z := t]}{x = t, \Gamma \vdash G[z := x]}$$

Figure 3: Additional rules for a proof-theoretic characterization of a rational-trees CLP language. A is a WRTA, and y is a new variable.

Again, we had only to give additional rules for the behaviour of closed arithmetic formulae; essentially, we have expanded the meaning of $=$ from identity between closed terms (the $t = t$ axiom) to arithmetic equality of closed terms (the $t = t$ axiom plus the new rules and axioms).

For theories like the ones given, the task of coming up with a proof-theoretic specification can be reduced to coming up with a proof-theoretic specification of the valid closed constraints. The essential property here is that the intended theory is *Henkin* [Sho67]; that is, whenever $\exists x B$ is true, there is a closed term t such that $B[x := t]$ is true².

Unfortunately, however, not all interesting constraint theories are Henkin. A simple example is the theory of rational trees: we have that $\exists x(x = f(x))$ is true, but there is no closed t such that $t = f(t)$. Another important example is the theory of real arithmetic, in which we have no closed term t such that $t \times t = 2$. For these theories, we have to use other methods.

4.1.2 Axiomatic Technique

Another technique for specification is to provide axioms which can be added to the left-hand side of a sequent in order to obtain proofs of solvable queries. Unlike the closed-constraint approach, this technique is universal.

Example: Rational trees. Van Emden and Lloyd [vEL84] and Maher [Mah88] have given Hilbert-style logical descriptions of the theory of rational trees, with varying soundness and completeness properties. The logical consequences of these theories include all formulae of the form

$$\exists x_1 \dots \exists x_n ((x_1 = t_1) \& \dots \& (x_n = t_n))$$

where the x_i 's are distinct and the t_i 's are terms whose variables are among the x_i 's. Let us call these the *weak rational tree axioms (WRTAs)*.

Using WRTAs, we can build a sequent-calculus characterization of a rational-trees CLP language. It consists of the rules in Figure 1 with the addition of the rules in Figure 3. An example derivation of $\exists x(x = f(f(x)))$ can be found in Figure 4.

□

²A more precise and traditional definition of Henkin theories is those theories T for which, for every sentence $\exists x A$ of the language, there is a constant e in the language such that $T \models (\exists x A) \supset (A[x := e])$.

$$\frac{\frac{\frac{\frac{y = f(y) \vdash f(f(y)) = f(f(y))}{y = f(y) \vdash f(y) = f(f(y))}}{\frac{y = f(y) \vdash y = f(f(y))}{\frac{y = f(y) \vdash \exists x(x = f(f(x)))}{\frac{\exists x(x = f(x)) \vdash \exists x(x = f(f(x)))}{\vdash \exists x(x = f(f(x)))}}}}}$$

Figure 4: Example derivation of $\exists x(x = f(f(x)))$ in the theory of rational trees.

Given any satisfiability function sat , we can develop a proof-theoretic characterization \vdash such that $\Gamma \vdash G$ if G succeeds with respect to Γ ; we can do this by giving rules which allow us to add any element of Θ_{sat} to the left-hand side. This \vdash does not necessarily have the converse property – that if $\Gamma \vdash G$ then G succeeds with respect to Γ – but as we will see later, we can achieve such results with the use of techniques from linear logic.

4.2 Proving Properties of Implementations

We are now in a position to tie together the theory of satisfiability functions with the techniques for description of CLP theories. Say we are given a satisfiability function sat on (\mathcal{L}, C) and a transition relation \rightarrow for sat . We are also given a proof system intended to describe the intended theory in a logical form. The kinds of results we would want to prove are:

- *Soundness of success*: if $\langle G, \emptyset \rangle \rightarrow^* \langle \emptyset, C \rangle$ w.r.t. program P , where $sat(C) \simeq true$, then $P \vdash G$.
- *Soundness of failure*: if $\langle G, \emptyset \rangle \rightarrow^*$ w.r.t. program P , then $P \not\vdash G$.
- *Completeness of success*: if $P \vdash G$, then $\langle G, \emptyset \rangle \rightarrow^* \langle \emptyset, C \rangle$ w.r.t. P , where $sat(C) \simeq true$.
- *Completeness of failure*: if $P \not\vdash G$, then $\langle G, \emptyset \rangle \rightarrow^* false$.

Clearly, if sat is strict and we can prove soundness and completeness of success, or soundness of success and failure, then we can prove the rest of the results.

I will just note here that the soundness and completeness of Prolog II with respect to the proof system given in the last subsection can be proven by techniques similar to those given in [Col83, Mah88, And89]. I will go into more detail about the CLP(R) example from Section 3.1.

Example: a characterization of CLP(R). Consider the satisfiability function sat and transition relation \rightarrow we defined for CLP(R) in Section 3.1. We can define a characterizing proof system for this operational semantics using the axiomatic technique and some ideas from linear logic [Gir87].

$$\begin{array}{c}
\frac{\Gamma_1, !\Gamma \vdash G_1 \quad \Gamma_2, !\Gamma \vdash G_2}{\Gamma_1, \Gamma_2, !\Gamma \vdash G_1 \& G_2} \quad \frac{\Gamma \vdash G_1}{\Gamma \vdash G_1 \vee G_2} \quad \frac{\Gamma \vdash G_2}{\Gamma \vdash G_1 \vee G_2} \\
\frac{\Gamma \vdash G[x := t]}{\Gamma \vdash \exists x(G)} \quad \frac{}{c, !\Gamma \vdash c} \\
\frac{!D, D, \Gamma \vdash G}{!D, \Gamma \vdash G} \quad \frac{D[x := t], \Gamma \vdash G}{\forall x D, \Gamma \vdash G} \quad \frac{\Gamma \vdash G}{(p(t_1, \dots, t_n) \leftarrow G), \Gamma \vdash p(t_1, \dots, t_n)} \\
\frac{\exists [c_1 \& \dots \& c_k], \Gamma \vdash G}{\Gamma \vdash G} \quad \frac{B[x := y], \Gamma \vdash G}{\exists x B, \Gamma \vdash G} \quad \frac{B, C, \Gamma \vdash G}{B \& C, \Gamma \vdash G}
\end{array}$$

Figure 5: A proof-theoretic characterization of CLP(R). Γ is any sequence of formulae, $!\Gamma$ is a sequence of formulae preceded by $!$, y is a new variable, and c_1, \dots, c_k are constraints such that $\text{sat}(\{c_1, \dots, c_k\}) \simeq \text{true}$.

The proof system is given in Figure 5. The important ideas in this proof system are as follows.

- As in the axiomatic characterization of rational trees, we allow axioms about satisfiable systems of equations to be introduced and manipulated in the antecedent (left-hand side) of a sequent.
- In the antecedent, we precede each rule from the program with the linear logic $!$ operator. This is an operator which allows for duplication of assumptions; for formulae in the antecedent not preceded by $!$, we do not allow duplication. (The antecedent is viewed as a multiset.)
- We require that at the top of the proof be axiomatic sequents of the form $c, !\Gamma \vdash c$, where $!\Gamma$ is a sequence of formulae *all* of which are preceded by $!$.
- We require that the sequence of formulae not preceded by $!$ be *split* across the two premisses of the rule introducing $\&$ on the right.

All this has the effect of forcing each assumption arising from the *sat* rule to be used once and only once in the course of a proof.

□

An example proof is given in Figure 6. Note that the proof would not have gone through if the query had been simply $\exists x, y(x \cdot y = 4)$, because we would have had the extra formula $y' = 2$ to contend with. Conversely, if we had allowed axioms to be simply of the form $(c, \Gamma \vdash c)$, we would have been able to prove $\exists x, y(x \cdot y = 4)$, even though that is not a linear equation and thus not solvable in the operational semantics. As it is, with respect to this proof system, we should be able to prove that the operational semantics has the properties of soundness and completeness of success and soundness of failure. Because *sat* is not strict in this case, however, we cannot prove completeness of failure.

This characterizing proof system is somewhat unwieldy due to the fact that the real-number axioms introduced in the antecedent may be huge. It may be

$$\begin{array}{c}
\frac{}{y' = 2, \forall z(p(z) \leftarrow z = 2) \vdash y' = 2} \\
\frac{}{y' = 2, \forall z(p(z) \leftarrow z = 2), (p(y') \leftarrow y' = 2) \vdash p(y')} \\
\frac{}{y' = 2, \forall z(p(z) \leftarrow z = 2), \forall z(p(z) \leftarrow z = 2) \vdash p(y')} \\
\hline
\frac{x' \cdot y' = 4 \vdash x' \cdot y' = 4}{x' \cdot y' = 4, y' = 2, \forall z(p(z) \leftarrow z = 2) \vdash x' \cdot y' = 4 \& p(y')} \\
\frac{x' \cdot y' = 4, y' = 2, \forall z(p(z) \leftarrow z = 2) \vdash \exists y(x' \cdot y = 4 \& p(y))}{x' \cdot y' = 4, y' = 2, \forall z(p(z) \leftarrow z = 2) \vdash \exists x, y(x \cdot y = 4 \& p(y))} \\
\frac{(x' \cdot y' = 4 \& y' = 2), \forall z(p(z) \leftarrow z = 2) \vdash \exists x, y(x \cdot y = 4 \& p(y))}{\exists y'(x' \cdot y' = 4 \& y' = 2), \forall z(p(z) \leftarrow z = 2) \vdash \exists x, y(x \cdot y = 4 \& p(y))} \\
\frac{\exists x', y'(x' \cdot y' = 4 \& y' = 2), \forall z(p(z) \leftarrow z = 2) \vdash \exists x, y(x \cdot y = 4 \& p(y))}{\forall z(p(z) \leftarrow z = 2) \vdash \exists x, y(x \cdot y = 4 \& p(y))}
\end{array}$$

Figure 6: An example proof in the characterizing proof system for CLP(R).

possible to simplify the axioms, but even as it stands the proof system can be used for proving properties of CLP(R) programs.

The linear logic technique may be useful for characterizing other incomplete CLP-like languages. Some implementations of finite-domain constraint solvers are incomplete for efficiency reasons [Mac85]; and many implementations of Prolog use unification without occurs check, which is sound with respect to the rational-tree axioms but which sometimes goes into an infinite loop.

5 Conclusions and Related Work

I have shown that the class of satisfiability functions adequately characterizes the behaviour of a wide variety of implementations of CLP languages, and that there is a simple, non-model-theoretic condition (“reliability”) for testing whether a satisfiability function is reasonable.

I have also discussed techniques for specifying and characterizing CLP systems with sequent calculi. I have pointed out that the question of whether the theory is Henkin is important, and that the notation and proof theory of linear logic (or other such “substructural” logics) can help in characterization.

The definition of a reliable satisfiability function is closely related to Scott’s definition of an *information system* [Sco82]. However, neither the space of satisfiability functions, nor the space of information systems (under a reasonable mapping from one notion to the other), are proper subsets of the other.

Höhfeld and Smolka [HS88] and Fröhwirth [Frü92] have both explored the idea of formally describing constraint theories. Höhfeld and Smolka describe an alternative framework to Jaffar and Lassez’s for constraint systems; like Jaffar and Lassez, however, they do not consider explicitly any computability restrictions on constraint satisfaction algorithms. Fröhwirth gives a Horn-clause-based language for defining constraint simplification rules, or SiRs, for any given domain. However, while SiRs have a logical form, they do not necessarily take the form of a simple and intuitive axiomatization or proof system.

There are several directions for future work in this area:

- Case studies. I would very much like to see these ideas applied for the purpose of fully and precisely characterizing existing, practical systems.
- Negation. I have avoided talking about negation in this paper because it poses general problems for logic programming theory which have not been adequately answered yet. A framework which characterizes the failure of constraint queries as well as their success would be desirable.
- Moving toward a standard description language. It would be premature at this point to propose some standard for describing constraint systems, but this would bring many benefits if done, much as BNF brought a standard manner of describing programming language syntax.

6 Acknowledgements

I appreciate the helpful comments and suggestions I have received from Veronica Dahl, Alistair Lachlan, Sanjeev Mahajan, Fred Popowich, Stephan Wehner (all of SFU), Thom Fröhwirth, Nevin Heintze, Tim Hickey, and Gert Smolka, as well as Torkel Franzén and the anonymous referees. This research has been supported by the Natural Sciences and Engineering Research Council of Canada, the SFU Centre for Systems Science, and the SFU President's Research Grant Committee, via Infrastructure Grants NSERC 06-4231, CSS 02-7960 and PRG 02-4028, Equipment Grants NSERC 06-4232, CSS 02-7961 and PRG 02-4029, Operating Grants OGP0002436 (Dahl) and OGP0041910 (Popowich), and the author's NSERC Postdoctoral Fellowship.

References

- [And89] James H. Andrews. Proof-theoretic characterisations of logic programming. In *Mathematical Foundations of Computer Science*, volume 379 of *Lecture Notes in Computer Science*, pages 145–154, Porąbka-Kozubnik, Poland, 1989. Springer.
- [Col83] Alain Colmerauer. Prolog and infinite trees. In K. L. Clark and S.-A. Tärnlund, editors, *Logic Programming*, pages 231–251. Academic Press, 1983.
- [Frü92] Thom Fröhwirth. Constraint simplification rules. Technical Report 92-18, ECRC, Munich, Germany, July 1992.
- [Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
- [HS84] Masami Hagiya and Takafumi Sakurai. Foundation of logic programming based on inductive definition. *New Generation Computing*, 2:59–77, 1984.

- [HS88] Markus Höhfeld and Gert Smolka. Definite relations over constraint languages. Technical Report 53, LILOG, IBM Deutschland, Stuttgart, Germany, October 1988. To appear in *Journal of Logic Programming*.
- [HSH90] Lars Hallnäs and Peter Schroeder-Heister. A proof-theoretic approach to logic programming I: Clauses as rules. *Journal of Logic and Computation*, 1(2), 1990.
- [JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the Conference on Principles of Programming Languages*, Munich, 1987.
- [JMSY92] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The CLP(\mathcal{R}) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, July 1992.
- [Kle52] Stephen Cole Kleene. *Introduction to Metamathematics*, volume 1 of *Bibliotheca Mathematica*. North-Holland, Amsterdam, 1952.
- [Mac85] Alan Mackworth. Constraint satisfaction. Technical Report 85-15, Department of Computer Science, University of British Columbia, September 1985.
- [Mah88] Michael J. Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. In *Proceedings of the Third Annual Symposium on Logic In Computer Science*, pages 348–357, Edinburgh, July 1988. Computer Society Press.
- [Mah93] Michael J. Maher. A logic programming view of CLP. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 737–753, Budapest, July 1993. MIT Press.
- [MNPS91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.
- [Mon76] James Donald Monk. *Mathematical Logic*, volume 37 of *Graduate texts in mathematics*. Springer-Verlag, New York, 1976.
- [Sco82] Dana Scott. Domains for denotational semantics. In *International Colloquium on Automata, Languages, and Programming*, 1982.
- [Sho67] Joseph Shoenfield. *Mathematical Logic*. Addison-Wesley, Reading, Mass., 1967.
- [vEL84] Maarten H. van Emden and John W. Lloyd. A logical reconstruction of Prolog II. *Journal of Logic Programming*, 2:143–149, 1984.

Programming with behaviors in an ML framework – The syntax and semantics of LCS

Bernard Berthomieu¹ and Thierry Le Sergent²

¹ LAAS/CNRS, 7 avenue du Colonel Roche, 31077 Toulouse Cedex, France.
e-mail: Bernard.Berthomieu@laas.fr

² LFCS, University of Edinburgh, The King's Buildings, Edinburgh, EH9 3JZ, UK.
e-mail: tls@dcs.ed.ac.uk (*On leave from LAAS/CNRS*)

Abstract. LCS is an experimental high level asynchronous parallel programming language primarily aimed at exploring design, implementation and use of programming languages based upon the behavioral paradigms introduced by CSP and CCS. The language extends Standard ML with primitives for concurrency and communication based upon a higher order extension of the CCS formalism. Typechecking enforces consistency of communications. An abstract operational semantics of the language is given in terms of a transition system.

1. Introduction

Motivations

A number of parallel programming languages are designed by extending sequential languages. Parallel programming facilities are provided through a set of primitives with a functional interface typically including a function for creating processes with some given text (e.g. fork), functions for implementing process communications and synchronization (e.g. channel, input, output, sync), and possibly some functions combining processes together (e.g. select). Examples of such languages are Poly/ML [12] and CML [17], both extending the language Standard ML.

Though concurrency and communications are provided through the use of functions, the semantics of these languages depart from the usual functional semantics due to the complex apparatus needed for casting nondeterminism and nonterminating computations into this paradigm. They are usually given a semantics in operational terms, by some reduction relation on programs of the language.

An alternative to this functional introduction of concurrency and communication is the behavioral approach promoted by CSP, CCS [13], and subsequent formalisms. The approach has been undoubtedly fruitful in the last decade in the areas of specification and analysis of concurrent systems. The central role in this paradigm is played by communications: a behavior describes a way of communicating with other behaviors, a program is some combination of communicating entities. The approach is inherently parallel, nondeterministic, and nonterminating systems are naturally expressed. The semantics of these languages are given in terms of some observation relation characterizing in some sense their communication capabilities.

Though the behavioral paradigm is often advocated as a convenient formalism for concurrency and communication, there has been very few work in designing programming languages that would canonically implement the paradigm. Noticeable exceptions are the language LOTOS [9], the language LCS introduced here and, to a lesser extend, the FACILE language [7].

Anatomy of LCS

LCS is an high level asynchronous parallel programming language primarily aimed at exploring design, implementation and use of programming languages based upon the behavioral

paradigm. The language and its implementations evolved through several versions [2] to that described here. A “sequentially running” implementation of LCS was made available to users in 1991 [4]; parallel and distributed implementations of the language are under way [10] [11].

Syntactically, LCS is designed as an extension of Standard ML [14]. Processes are described in an ML framework, but parallel programming capabilities are provided through a specific sub-language of behavior expressions rather than as a set of functions. Behavior expressions describe processes and evaluate to behavior values; behavior values may be turned into processes by specific process creation commands. A non trivial extension of the ML static polymorphic discipline enforces consistency of communications.

The language of behavior expressions is based on Robin Milner’s Calculus of Communicating Systems [13], which is widely accepted as a convenient formalism for concurrency and communication. However, CCS, as a bare abstract formalism, lacks several ingredients for constituting a convenient programming language (e.g. how to express usual values) and also suffers some known deficiencies in terms of expressiveness. In particular, CCS is a first order formalism with respect to behaviors and to communication labels, and its set of behavior combinators may lack the flexibility required for comfortably programming real applications. Finally, CCS does not consider at all assignable values and side-effects. LCS proposes solutions to all these problems that, as far as possible, preserve the underlying theory of CCS and follow the SML design principles as well.

Organization of the paper

Section 2 introduces the features of LCS and shows its embedding into Standard ML. Section 3 describes its original typing technique for behaviors. The operational semantics of LCS programs is detailed in Section 4 in terms of a transition system and an observation relation. Section 5 briefly overviews implementation. The concluding section summarizes the experience and compares it with related projects. It is assumed throughout that the reader has some knowledge of the essential features of both the language ML and the CCS formalism.

2. Syntax and features of LCS

Syntax

The language of expressions extends that of Standard ML with constructs for building behaviors. The essentials of the syntax of LCS is given in Table 2.1. The set of ML types is also enriched with “behavior types”; the typing aspects will be discussed in Section 3. In addition to the capabilities of an SML implementation, LCS implementations have commands for starting processes to run in the foreground or background in parallel with all other running processes.

Behaviors must not be confused with processes. Evaluation of a behavior expression does not produce a process, but rather a closure similar to a function value. Creating a process from a behavior value by one of the process creation commands causes the recursive evaluation of the body of the behavior. The operational semantics of LCS processes is detailed in Section 4.

Behaviors as communication capabilities

Behaviors must be understood as interaction capabilities. The behavior constructs of LCS strictly include those of CCS. Expression **stop** denotes the behavior that has no communication capability; behaviors are built from **stop**, possibly recursively, by action prefixing, compositions, restrictions and relabellings.

<code>exp ::= <sml expression> beh</code>	
<code>beh ::= stop</code>	null behavior
<code>{do e} => exp</code>	commitment action
<code>port ? {pat} => exp</code>	input action
<code>port ! {exp} => exp</code>	output action
<code>signal exp {with exp}</code>	event action{with message}
<code>exp C exp</code>	compositions
<code>exp catch erules</code>	event handling
<code>exp "{/lab₁ ... lab_n}"</code>	label restriction
<code>exp "{lab₁' ... lab_n'/lab₁ ... lab_n}"</code>	label relabelling
<code>C ::= & /& &/ /&/</code>	parallel compositions
<code>\& \&/ &\& \&/ \&/\&</code>	choice compositions
<code>erules ::= exp with match { exp with match}</code>	event handler rules
<code>port ::= label {# exp}</code>	communication ports
<code>match ::= pat => exp { match}</code>	SML matches
<code>pat ::= <sml pattern></code>	SML patterns

Table 2.1. LCS expressions

The construct “=>” prepends an action to a behavior. Actions may be either that of performing an internal move (the τ action of CCS), that of proposing a value on a communication port (output or signal), or that of accepting a value on a port (input or event handling).

There are three basic forms of compositions: parallel ($/\backslash$), choice ($\backslash/$) and **catch**; the first two are inherited from CCS. Processes composed in parallel may communicate values. The single communication and synchronization primitive is rendez-vous over named ports. Choice composition implements a selection mechanism, the first process in a choice composition that performs an action makes the alternative process(es) in the composition terminate. The last kind of composition (catch) is a particular parallel composition with the additional effect of terminating the behavior on the left side of catch as soon as a communication has occurred between the members of the composition. The signal and catch expressions implement a process-level exception mechanism that will be shortly described in more detail.

As in the CHOCS calculus [19], the class of behaviors and the class of messages in LCS are the same syntactic class. One can freely pass behavior values (not processes) as messages, or declare functions taking behaviors as arguments and/or returning behaviors as results.

Communication ports

The basic CCS formalism does not allow communication links to be passed between behaviors. Richer formalisms have been proposed that provide such a facility [6] [15]. LCS settled for a more conservative approach based on parametric channels as introduced in [1]; this permits to retain most of the underlying theory of CCS. LCS offers to compute communication ports, in some sense, instead of offering passing labels or channels between behaviors.

Ports in LCS have two components: a label in the class of identifiers, and an extension. Extensions may be any expression denoting a value of a type admitting equality (behavior values, like function values, do not admit equality; event values must). All ports have an extension, the default extension is the value “()” of type unit. Beside their extensions, the treatment of

communication ports is that of CCS: ports do not denote values and have global scope unless otherwise mentioned, in contrast with the treatment retained in PFL [8] or FACILE [7].

Operationally, port labels are implicitly bound to communication lines (one for each possible extension of the label) by the enclosing context. The scope of labels is controlled with restrictions. The restriction of p in b (written $b\{/\!p\}$) has the effect of delimiting the scope of ports labelled p occurring within b to expression b (as the lambda-abstraction in $\lambda p.b$ delimits the scope of the inner p to expression b). Relabellings help to build connected systems of behaviors; the relabelling $(b\{p/q\})$ restricts label p and makes ports labelled q within b appear to the enclosing context as having label p . The restrictive effects of LCS relabellings make them slightly different from CCS relabellings.

Labels may be hidden or renamed, but may not be computed nor passed as messages or parameters. On the other hand, extensions may be computed or passed, but may not appear in restrictions or relabellings; these apply to all ports with the labels involved, collectively. This treatment allows one to compute communication ports (their extensions, actually), while preserving the possibility of static typechecking.

Beside permitting a form of mobility [15], port extensions are particularly convenient for implementing systems constituted of arrays of processes in which the behavior of the individual processes can be parameterized by their index in the structure such as neural or systolic systems.

A simple example

The following behavior offers all prime numbers on a port labelled `out` by the Eratosthenes sieve method. `primes` is built from a behavior `succs` that offers all integers greater than its parameter on a port `out` (with the trivial extension), a behavior `filter` that echoes the integer it reads except those which are multiple of its parameter, an adhoc infix piping combinator (\wedge) and the recursive `sieve` behavior. On output of each prime number, behavior `sieve` evolves to itself preceded by a filter behavior that absorbs all multiples of that prime number.

```

val primes = (* offers prime numbers on port out *)
  let fun a  $\wedge$  b = (a {tmp/out})  $\wedge\!$  b {tmp/inp}) {/tmp}
    fun succs n = out! n=> succs (n+1)
    fun filter p = inp? x=>
      if x mod p = 0 then filter p else out! x=> filter p
    val rec sieve = inp? x=> out! x=> (filter x  $\wedge$  sieve)
    in succs 2  $\wedge$  sieve
  end;

```

Events, an exception mechanism at process level

Events, and the related event raising (**signal**) and event trapping (**catch**) expressions, implement an exception mechanism at process level. The exceptions we are interested in raising and trapping here are those related with communications such as the inability of some process to provide an offer on some communication port. Such exceptions would typically be handled by a reconfiguration of all or part of the current system of processes.

Events implement a particular kind of communication ports, obeying their own scope rules. Events are built from event constructors declared through specific declarations. Event declarations possibly make explicit a parameter type and a message type associated with the constructors. Since events must be comparable for equality, their parameter, if any, must admit equality in the SML sense. The message type associated with the constructor is the type of messages transmitted to the handlers when events built with that constructor are trapped.

Raising an event by **signal** offers the message associated with the event on a communication port identified by the event itself. Handling an event by **catch** must be understood as a communication between the process that offers the event and the handler. In addition to passing a message, this communication has the effect of terminating all processes under the catch.

Event constructors follow the same static scoping rules than other constructors, but the communication ports defined by events have particular scope rules. An event can only be caught by a handler if issued from the behavior on the left side of the catch composition. Further, the scope of an event stops at the innermost handler for that event. That way, events propagate like exceptions in SML, but event handling differs from exception handling in that signalling an event has no effect when no surrounding handler may catch it.

Beside their use as an exception mechanism, events are also useful as a clean-up mechanism to abort, on completion on some work, all the possibly awaiting processes that have been created as part of this work. The next example shows for instance how the previous prime behavior could be used to create an other behavior that offers on port `out` the nth prime number. Behavior `nthp` spawns primes, skips all primes up to the nth, and then raises a `BYE` event carrying that nth prime as message. The handler for event `BYE` offers its message on `out`; handling the event has the effect of aborting all processes resulting from execution of `primes`.

```
fun nthp n = (* offers nth prime on out *)
let event BYE with int
    fun skip n = out? x=>
        if n>1 then skip (n-1) else signal BYE with x
    in (primes /\\ skip n) {/out} catch BYE with p => out! p=> stop
end;
```

Side effects, imperative features

Any evaluation may have a side-effect. Reference values are handled as in SML, but memory locations have restricted scopes here. Each running instance of a behavior (i.e., each process) is associated with a store; side-effects in evaluation of a process are restricted in scope to the store associated with it. That store may be private, or shared with some other process(es).

The basic store assignment mechanism for processes is inheritance. When started, a process inherits the store of the toplevel process or a copy of it depending on the process creation command used. The stores of processes created from expansions of running processes are determined by the composition combinators. The light compositions (`/\` and `\/\`) create processes that both inherit the current store and share it. With left strong compositions (`//\` and `\\\`), the process on the left hand side inherits a copy of the current store, while the process on the rhs inherits the current store. The other composition operators can be derived from those (see their semantics in Chapter 4). The **catch** construct acts as a light composition; strong catch compositions can easily be simulated with the existing composition operators, if required.

An example of use of strong compositions is provided by the implementation of the LCS interactive toplevel. User environments are conveniently kept in references, updated at each toplevel declaration; the LCS toplevel itself is implemented as a system of LCS processes sharing the same store. Creating a multi-user implementation of LCS is which all users having successively logged in get the same initial environment, but do not share its further modifications is naturally implemented with strong compositions.

For reasons that will be explained in Section 4, port extensions and messages are not allowed to hold or encapsulate reference values. Current implementations, however, do not enforce this constraint; both static and dynamic methods are being investigated for that purpose.

User interface

In addition to the toplevel commands provided by SML user interfaces (declarations), LCS interactive user interfaces offer commands for process management.

Process creation commands create processes to run in parallel with all other currently running processes (including the toplevel). As processes successively started may have to communicate, it is essential that these processes agree on the types of the ports they use to communicate. This typing constraint is enforced when processes are started, rather than when declared. In other words, communication ports are not bound to actual communication lines at compile-time, but are dynamically bound when processes are started. For implementing the type constraint, the LCS toplevel maintains a type information that, at any time, holds the type of the parallel composition of all processes started from the beginning of the session or the last clean-up command for user processes. This type is updated by process creation commands and reset by the process clean-up command.

Commands **start** and **Istart**, taking a behavior expression as parameter, create processes to run in background. They return immediately. The former assigns to the process a private store consisting of a copy of the store of the toplevel process; processes started by the latter share their store with that of the toplevel process. It is occasionally useful to run processes in foreground rather than background (e.g. when the user process receives input from the same window that the toplevel). Foreground running capabilities are provided by commands **call** and **Icall**, which create processes with private or shared storage, respectively, together with a specific **return** behavior, semantically equivalent to **stop**. Evaluation of **return** makes the corresponding call command return (evaluation of the called process resumes in background).

Finally, a **kill** command allows the user to aborts all the processes started, and a **reboot** command permits to restart the LCS runtime on any user provided behavior; this last command is used for bootstrapping new versions of the compiler and building stand-alone applications.

3. Typing behavior expressions

The typing rules for behavior expressions must enforce type-safe communications between processes. A simple rule ensures this: whenever their scopes intersect, ports with the same label must transmit values of the same type and must have extensions of the same type. The rule is sufficient though not necessary; it is only necessary for pairs of matching ports which may actually be involved in a communication, but, obviously, this cannot be generally inferred at compile-time. In addition, port extensions must be restricted to values of types admitting equality in the Standard ML sense since comparing extensions for equality is needed to determine actual communications capabilities at run time.

In languages in which communication channels are values, such as PLF or CML, channels are given a type (α *chan*), in which α is the type of the values the channel may pass, and processes are given a trivial constant type. A distinct approach, suitable in languages obeying the behavioral paradigm, such as LCS, is to assign to processes types that tell, for each channel they may use for communication, the type of items that may be passed through this channel. In addition, LCS behavior types will assign to each such label the type of its extension. Now, higher order prevents to compute the exact set of ports through which some behavior may communicate; behaviors will thus be assigned types which make precise an extension and a message type for every possible communication label.

Syntactically, LCS behavior types are built from special type variables called behavior type variables and written $_a$, $_b$, etc., possibly prefixed by a finite number of fields of the form {label: extension_type # message_type} where extension_type is an equality type. Regular type variables (' a ', ' b ', etc.) may be substituted by any type, including behavior types, but behavior type variables may only be substituted by other behavior type variables or behavior types. Behavior types thus constitute a strict subset of all types.

Semantically, behavior types may be read as total functions assigning types to labels. A behavior type variable assigns to every possible label a type ' $'a\#b$ ' with variables ' $'a$ ' and ' $'b$ ' not occurring elsewhere (' $'a$ is an equality type variable). The type $\{p:\tau\}p$ denotes the function that associates type τ to label p , and type $(p\ q)$ to labels q distinct from p . Unconstrained behavior types sharing no variables are denoted by distinct behavior type variables.

As an example, the type $\{q:\text{bool}\#\text{int}, r:\text{unit}\#('a*\text{bool})\}_b$, abbreviated $\{q.\text{bool}:\text{int}, r:'a*\text{bool}\}_b$, is the most general type for behaviors that may send or receive integers through ports labelled q indexed by booleans and pairs of type ' $a*\text{bool}$ ' through ports labelled r with the default extension. Its unification with the type $\{s:\text{int}, r:'a->\text{int}*'b\}_c$ yields the type $\{q.\text{bool}:\text{int}, r:'a->\text{int}*\text{bool}, s:\text{int}\}_d$.

Behavior types are conveniently formalized as a particular subset of the "Tagged Types" introduced in [3] where their theory, including canonical forms and unification algorithms is developed in depth. Unification of Tagged Types is reducible to standard unification for first order terms. Conceptually, tagged types bear strong relationships with Wand's row types [20] and Rémy's record types [18], but the logical and technical treatments of tagged types are original and yield, we believe, a simpler and more intuitive treatment.

The typing rules for behavior expressions are summarized in Table 3.1; other expressions of the language are typed as in SML. Events receive type $(\text{ty} \text{ evt})$ where ty is the type of the message brought by the event.

expression	is typed as	where
stop	Stop	Stop : $_a$
$=>e$	Tau e	Tau : $_a -> _a$
$p\#e?x=>e'$	Input _p e ($\lambda x.e'$)	Input _p : " $a -> (b -> \{p.'a: b\}_c) -> \{p.'a: b\}_c$ "
$p\#e!e'=>e''$	Output _p e e' e''	Output _p : " $a -> b -> \{p.'a: b\}_c -> \{p.'a: b\}_c$ "
$e\{p\}$	Hide _p e	Hide _p : $\{p.'a: b\}_c -> \{p.'d: e\}_c$
$e\{q/p\}$	Rename _{q,p} e	Rename _{q,p} : $\{p.'a: b, q.'c: d\}_e -> \{p.'f: g, q.'a: b\}_e$
$e \circ e'$	Compose (e,e')	Compose : $_a * _a -> _a$
$A \vdash e : \text{ty} \text{ evt}$	$A \vdash e' : \text{ty}$	$A \vdash e_i : \text{ty}_i \text{ evt} \quad A \vdash e'_i : \text{ty}_i \quad A \vdash e : _b \quad A \vdash h_i : _b$
$A \vdash \text{signal } e \text{ with } e' : _b$		$A \vdash e \text{ catch } e_1 \text{ with } e'_1 => h_1 \parallel \dots \parallel e_n \text{ with } e'_n => h_n : _b$

Table 3.1. Type inference for behavior expressions

As an example, the "pipe" combinator (^) defined in the body of behavior primes in Section 2 would receive as principal type:

$$\wedge : \{\text{out}.'a: b\}_c * \{\text{inp}.'a: b, \text{tmp}.'d: e\}_c -> \{\text{tmp}.'f: g\}_c$$

Which can also be written in another, more verbose, canonical form:

$$\begin{aligned} \wedge : & \{\text{inp}.'a: b, \text{out}.'c: d, \text{tmp}.'e: f\}_g * \{\text{inp}.'c: d, \text{out}.'h: i, \text{tmp}.'j: k\}_g \\ & \rightarrow \{\text{inp}.'a: b, \text{out}.'h: i, \text{tmp}.'l: m\}_g \end{aligned}$$

It can be seen from this type that port `inp` (resp. `out`) has in the type of expression $(A \wedge B)$ the type that port `inp` has in A (resp. port `out` has in B).

4. Operational semantics

Core formalism and translation

The operational semantics of LCS is obtained from the semantics of a richer “core” language. Expressions e of this core language include lambda-expressions, with a recursion combinator, the usual constants, primitives for references, and the following behavior expressions:

$b ::= \text{stop} \mid \tau.e \mid p(e_1)!e_2.e_3 \mid p(e_1)?x.e_2 \mid e\text{E} \mid e[S] \mid e_1|e_2 \mid e_1+e_2 \mid e_1 < e_2 \mid <e>$

p, q, r, \dots , etc, are labels in some denumerable set Σ .

This core language extends CCS in several ways: lambda abstraction and application are primitive; behaviors may be passed as messages between processes; communication labels are parameterized; there is a new binary combinator (written $<$ and read “interrupt”) and, finally, there is a specific construction for management of stores: $< >$. We shall not precisely give here a concrete syntax for restriction and relabelling expressions, but simply assume that these expressions may at least denote restriction or relabelling of either a single port or of all ports bearing some given label. Port extensions are restricted to some subset of expressions admitting canonical forms (corresponding to the expressions having equality types).

The translation of LCS expressions into core expressions is summarized in Table 4.1. $\lceil \cdot \rceil$ is the translation function. The translation of functional expressions is standard and has been omitted. The expression “**signal** e with m ” is interpreted as an output offer of message m on a port made from a label χ , assumed not part of Σ , with extension e . The sequence of handlers for the different events is interpreted as the disjunction of the handlers, each guarded by an input on a port labelled ζ (with $\zeta \notin \Sigma$) with the corresponding event expression as extension. It results from the translation of the catch construct that the dynamic scope of an LCS event extends no further than its innermost handler. The effects of the other composition operators are obtained from a combined use of the core operators $|$, $+$ and $< >$.

$\lceil \{\text{do } e\} \Rightarrow e' \rceil = \tau. \lceil (\{e\}; e') \rceil$	$\lceil \text{signal } e \text{ with } e' \rceil = \chi(\lceil e \rceil)! \lceil e' \rceil. \text{stop}$
$\lceil p\#e! e' \Rightarrow e'' \rceil = p(\lceil e \rceil)! \lceil e' \rceil. \lceil e'' \rceil$	$\lceil e \{/\text{p}\} \rceil = \lceil e \rceil \backslash p$
$\lceil p\#e? x \Rightarrow e'' \rceil = p(\lceil e \rceil)? x. \lceil e'' \rceil$	$\lceil e \{q/p\} \rceil = (\lceil e \rceil \backslash q) [q/p]$
$\lceil e \text{ catch } e_1 \text{ with } m_1 \Rightarrow h_1.m_1 \parallel \dots \parallel e_n \text{ with } m_n \Rightarrow h_n.m_n \rceil =$	
$(\lambda x_1. \dots. \lambda x_n. \lceil e \rceil [\zeta(x_1)/\chi(x_1)] \dots [\zeta(x_n)/\chi(x_n)]$	
$< (\zeta(x_1)? m_1. \lceil h_1.m_1 \rceil + \dots + \zeta(x_n)? m_n. \lceil h_n.m_n \rceil) \backslash \zeta) \lceil e_1 \rceil \dots \lceil e_n \rceil$	
where x_i assumed not to occur free in h_j or e , and $\chi, \zeta \notin \Sigma$.	
$\lceil e \wedge e' \rceil = \lceil e \rceil \lceil e' \rceil$	<i>light compositions</i>
$\lceil e \wedge\!\! e' \rceil = \lceil e \rceil \lceil e' \rceil \triangleright$	<i>right-strong compositions</i>
$\lceil e / \wedge e' \rceil = \lceil e' \wedge e \rceil$	<i>left-strong compositions</i>
$\lceil e / \wedge\!\! e' \rceil = \lceil (\text{stop} \wedge e) \wedge e' \rceil$	<i>strong compositions</i>
$\lceil e \vee e' \rceil = \lceil e \rceil + \lceil e' \rceil$	
$\lceil e \vee\!\! e' \rceil = \lceil e \rceil + \lceil e' \rceil \triangleright$	
$\lceil e \vee\!\! e' \rceil = \lceil e' \vee e \rceil$	
$\lceil e \vee\!\! e' \rceil = \lceil (\text{stop} \vee e) \vee e' \rceil$	

Table 4.1. Translation into core formalism

The semantics discussed in the next sections is restricted to the subset of terms of the core language which can result from translation of well-typed LCS expressions.

Expressions, Values, Reduction (\rightarrow)

Evaluation in LCS is “applicative”. Evaluating a behavior in operand position, for instance, produces a value, but we do not want compositions or communications occurring within that behavior to be evaluated at that time. This first evaluation mechanism is called *reduction*.

The *reduction* relation, written \rightarrow , is the usual “evaluation” relation for functional expressions. The expressions which are irreducible by \rightarrow are the *values*. Reduction is defined in Table 4.2 in which e, e' range over expressions, x, x' range over identifiers, v, v' , range over values and $e\{v/x\}$ is the expression obtained by substituting v for the free occurrences of x in e (including within restriction and relabelling expressions).

Expressions are reduced in the context of a store, which may be updated as the result of reductions. For reasons that will shortly appear, we shall use the word *segment* here, in place of store. A segment maps *locations* (addresses) to values. In Table 4.2, s, s' range over segments, a, a' range over locations and $s+(a,v)$ is the segment obtained from s by adding the new (location,value) pair (a,v) .

$e_1, s \rightarrow e_1', s'$		$\text{rec } x.e, s \rightarrow e \{ \text{rec } x.e/x \}, s$	(r4)
$e_1 e_2, s \rightarrow e_1' e_2, s'$	(r1)	$e_1, s \rightarrow e_1', s'$	
$e_2, s \rightarrow e_2', s'$	(r2)	$p(e_1)!e_2.e_3, s \rightarrow p(e_1')!e_2.e_3, s'$	(r5)
$v e_2, s \rightarrow v e_2', s'$		$e_1, s \rightarrow e_1', s'$	
$(\lambda x.e) v, s \rightarrow e \{ v/x \}, s$	(r3)	$p(e_1)?x.e_2, s \rightarrow p(e_1')?x.e_2, s'$	(r6)
$a \notin \text{dom}(s)$	(r7)	$\text{deref } a, s \rightarrow (s a), s$	(r8)
$\text{ref } v, s \rightarrow a, s+(a, v)$		$a := v, s \rightarrow (), s+(a, v)$	(r9)

Table 4.2. The reduction relation \rightarrow

Rules (r1) to (r4) are standard; they define an applicative order evaluation. In (r4), it is assumed that e is either a lambda abstraction or a behavior expression (this is enforced by syntactical constraints in LCS). Evaluation of input (r6) or output (r7) constructs consists of evaluating their communication port extensions. Constants, lambda expressions and other behavior expressions are irreducible by \rightarrow . Rules (r7) to (r9) concerning the imperative aspects are also standard.

Threads, Suspensions, Expansion (\Rightarrow)

Creating a process from a behavior should force the recursive evaluation of its content, including communications and creations of other processes. This second evaluation mechanism will be called *expansion*, since it generally results in creating processes. Expansion builds a structured system of threads from a behavior expression.

The *expansion* relation, written \Rightarrow , relates *states* constituted of a thread (also called a *process*, indifferently) and a *store*. Threads irreducible by expansion are called *suspensions*.

A thread is a simple thread, or a thread built from another by a thread restriction or relabelling operator $\{\backslash, /\}$, or a compound thread built by one of the thread composition operators $\{\parallel, ++, <<\}$.

Simple threads may operate either on private or on shared segments. The structure linking segments together is the *store*. Stores map segment identifiers to segments, each segment mapping locations to values. For some store S and segment identifier g, (S g) is a segment and ((S g) a) is the content of location a in segment (S g).

Simple threads are pairs $\langle e, g \rangle$, where e is an expression typing as a behavior, and g is a segment identifier. A state is a pair (b, S) , where b is a (possibly compound) thread, and S is a store. A *thread context* $C[\]$ is a possibly compound thread expression in which a simple thread is replaced by a hole; $C[t]$ is the thread obtained by filling the hole of $C[\]$ with thread t.

The expansion relation is defined in Table 4.3 in which b, b' range over threads, S, S' range over stores, s, s' range over segments, g, g' range over segment identifiers and $S+(g, s)$ is the store obtained from S by binding the segment s to the identifier g.

$b, S \ g \rightarrow b', s'$		$\langle e \setminus E, g \rangle, S \Rightarrow \langle e, g \rangle // E, S$	(e4)
$\langle b, g \rangle, S \Rightarrow \langle b', g \rangle, S+(g, s')$	(e1)	$\langle e [R], g \rangle, S \Rightarrow \langle e, g \rangle // R, S$	(e5)
$e_1, S \ g \rightarrow e_1', s'$		$\langle e_1 e_2, g \rangle, S \Rightarrow \langle e_1, g \rangle // \langle e_2, g \rangle, S$	(e6)
$\langle p \# v! e_1.e_2, g \rangle, S \Rightarrow \langle p \# v! e_1'.e_2, g \rangle, S+(g, s')$	(e2)	$\langle e_1 + e_2, g \rangle, S \Rightarrow \langle e_1, g \rangle ++ \langle e_2, g \rangle, S$	(e7)
$g' \notin \text{dom}(S)$	(e3)	$\langle e_1 < e_2, g \rangle, S \Rightarrow \langle e_1, g \rangle << \langle e_2, g \rangle, S$	(e8)
		$b, S \Rightarrow b', S'$	
$\langle \langle e \rangle, g \rangle, S \Rightarrow \langle e, g' \rangle, S+(g', S g)$		$C[b], S \Rightarrow C[b'], S'$	(e9)

Table 4.3. The expansion relation \Rightarrow

Rules (e1) and (e2) link the reduction and expansion relations; expanding an expression requires its reduction. Rule (e2) expresses that messages must be reduced prior to sending them. Rules (e4) to (e8) define the expansion of restriction, relabelling and compositions operators, which build structured threads. The binary composition combinators produce two simple threads from the current one, each inheriting the current segment identifier. Rule (e3) explains the segment copying effects of construction $\langle \rangle$. Finally, rule (e9) defines the expansion of compound threads from the expansions of its constituent simple threads.

Communication and pruning ($\equiv_{\mu} \Rightarrow$)

Finally, evaluation of a process involves communications. The *communication* relation, written $\equiv_{\mu} \Rightarrow$, also relates states. It is defined from a family of relations indexes by *actions*. The treatment below is adapted from the classical “observation” relation for CCS.

An action is either the particular action τ , or a triple (l, Δ, v) , where l is a port value, or *line*, consisting of a label in $\Sigma \cup \{\chi, \zeta\}$ and a reference-free value admitting a canonical form (a port index), Δ is a *direction* (! or ?), and v is a *message* constituted of a reference-free value.

The communication relation is defined in Table 4.4 in which S, S' range over stores, b, b' range over threads and w, w' range over suspensions. Arbitrary actions are ranged over by μ , μ' . Given an action μ , $\bar{\mu}$ denotes the action with the opposite direction, but same line and message fields than μ . For conciseness, rules sharing the same premises are grouped into single rules with a multiple denominator.

Rules (c1) to (c3) define the atomic actions. The other rules, except (c15) to (c18) express observation of actions in the different possible contexts, rules (c15) and (c16) express commu-

$\langle t.e,g \rangle, S \equiv t \Rightarrow \langle e,g \rangle, S$	(c1)	$w_1, S \equiv \mu \Rightarrow b_1, S$	
$\langle p(v) ! v'.e,g \rangle, S \equiv (p(v), !, v') \Rightarrow \langle e,g \rangle, S$	(c2)	$w_1 ++ w_2, S \equiv \mu \Rightarrow b_1, S$	(c9)
$\langle p(v)?x.e,g \rangle, S \equiv (p(v), ?, v') \Rightarrow \langle e\{v' / x\}, g \rangle, S$	(c3)	$w_2 ++ w_1, S \equiv \mu \Rightarrow b_1, S$	(c10)
$w, S \equiv t \Rightarrow b, S$	(c4)	$w_1 w_2, S \equiv \mu \Rightarrow b_1 w_2, S$	(c11)
$w // E, S \equiv t \Rightarrow b // p, S$		$w_2 w_1, S \equiv \mu \Rightarrow w_2 b_1, S$	(c12)
$w, S \equiv (l, \Delta, v) \Rightarrow b, S$	$l \notin E$	$w_1 << w_2, S \equiv \mu \Rightarrow b_1 << w_2, S$	(c13)
$w // E, S \equiv (l, \Delta, v) \Rightarrow b // E, S$		$w_2 << w_1, S \equiv \mu \Rightarrow b_1, S$	(c14)
$w, S \equiv t \Rightarrow b, S$	(c6)	$w_1, S \equiv \mu \Rightarrow b_1, S \quad w_2, S \equiv \bar{\mu} \Rightarrow b_2, S$	
$w // R, S \equiv t \Rightarrow b // R, S$		$w_1 w_2, S \equiv t \Rightarrow b_1 b_2, S$	(c15)
$w, S \equiv (l, \Delta, v) \Rightarrow b, S$	$l \notin \text{dom}(R)$	$w_1 << w_2, S \equiv t \Rightarrow b_2, S$	(c16)
$w // R, S \equiv (l, \Delta, v) \Rightarrow b // R, S$		$b, S \Rightarrow w, S' \quad w, S' \equiv \mu \Rightarrow b, S''$	(c17)
$w, S \equiv (l, \Delta, v) \Rightarrow b, S$	$l' = R l$	$b, S \equiv \mu \Rightarrow b, S' \quad b, S' \Rightarrow b', S''$	(c18)
$w // R, S \equiv (l', \Delta, v) \Rightarrow b // R, S$		$w, S \equiv \mu \Rightarrow b', S''$	

Table 4.4. The communication relation $\equiv \mu \Rightarrow$

nications between processes composed by \parallel and $<<$, respectively. The communication rule for \parallel is exactly that of the $|$ combinator of CCS. Except for the interaction rule (c16), the rules for interruption $<<$ resemble those of the “disable” combinator of the language LOTOS [9]. For behaviors resulting from translation of LCS expressions, a communication involving the component on the right side of combinator $<<$ necessarily involves the component on its left side. The translation of signal and catch constructions imply that events are trapped by their innermost handler, if any is provided. The last two rules relate expansion with communication.

Operationally, a communication has two simultaneous effects. The first is that of passing a value from one thread to another (rules (c15) and (c16)). The other effect is that of pruning the current thread by removing some of its components (rules (c9), (c10), (c14) and (c16)).

These rules extend the CCS and CHOCS calculus. Compared to CCS, and besides stores and segments, the values in actions may include behavior values, ports have a richer structure (label plus extension), and the interrupt rules are added.

Reference passing

Neither port extensions, nor messages, may encapsulate reference values. The reasons for these restrictions are briefly explained here.

First, it seems natural that the locations created in some segment are not defined in the other segments. Allowing to pass locations would thus require to interpret dereferencing of a location not belonging to the segment of the process.

Next, an important question is whether a copy of a location (obtained by segment copying) should be considered an equal or a different location than the original. Considering them different (while allowing reference passing) would be semantically satisfying, but would lead to severe implementation problems since it would require to duplicate the whole environment of

a thread upon a strong composition. On the other side, considering them the same would only require to copy the segment of the thread, what can be efficiently implemented by a simple copy-on-write technique, but location passing would then have a few counter-intuitive effects. In particular, the content of some location could change when passed from a process to another, and ports extensions or events could be considered equal though constituted of references with different contents (when compared for equality for determining communication capabilities).

The solution retained makes LCS a conservative extension of Standard ML with respect to the treatment of references (assuming there is a single thread running, or that only strong compositions are used), it also permit to avoid the effects mentioned and allow an efficient implementation. Finally, note that location passing would not cause any problem if all programs shared the same segment, but this was found too severe a restriction.

Semantics

Having defined an observation relation in the style of what has been done for CCS, we would like to take as semantics of LCS programs the observation congruence discussed in [13], or a congruence included in it such as the strong equivalence. Artesiano and Zucca [1] have shown that adding indexed communication ports to CCS could be easily accommodated by its observational semantics. Thomsen [19] observed that the notion of observation congruence could be easily extended to handle behavior passing. Furthermore, the adjunction of the interrupt combinator would not bring any specific difficulty. So, not taking assignments into account, a suitable concept of observation congruence can be formalized for the core calculus of LCS.

However, this concept of observation congruence would not be adequate for arbitrary LCS programs. In general, the communication actions themselves and/or the way they are organized may depend on the (non observable) expansions preceding the communications. Consider for instance the following behavior a, parameterized by some integer k>0:

```
fun a k =
  let val r = ref 0
  in (while deref r < k do r := deref r + 1; stop) /\ 
    let val x = deref r in p#x! => stop end
  end;
```

The behavior (a K), for some integer K, may offer an output on any line p#i, for i in the range 0...K-1. The port on which a value is offered only depends on the number of loops performed while reducing the do-while expression in the left side of the parallel composition before the computation of x on the right side has been completed. Observation congruence should not identify this program with the following:

```
fun b k = p#0! => stop /\ ... /\ p#(k-1)! => stop;
```

Which is clearly distinct from the former since only one possible communication offer is provided by (a K), while all alternatives are simultaneously offered by behavior (b K).

So, not surprisingly, side-effects and shared segments together bring a kind of nondeterminism which is not properly captured by the observation congruence concept. This nondeterminism was observed earlier by researchers concerned with verification of parallel programs; a frequent assumption required in their proof techniques was that programs are “interference-free” (see for instance [16]). However, observation congruence is a suitable semantics for the class of interference-free LCS programs. How to determine the property for a given program is a complex task in general, but, fortunately, there is a large class of structurally interference-free programs: those only using strong compositions. It can be shown that the programs obeying this restriction cannot exhibit the particular nondeterminism discussed above.

5. An overview of implementations

Abstract machines

The abstract machine associates a virtual processor with every active thread (corresponding to the non-suspension simple threads discussed in Section 4). These processors reduce and expand threads asynchronously and cooperate for communication.

The state (registers) of each thread include registers for reduction, a communication environment linking communication ports to “communication lines”, a “position” register for implementing process preemption and a store segment identifier. The global state information is constituted of all active threads, the content of the communication lines (the lines may be shared by several threads), a shared information called the accumulator and the content of all segments of the store, all soon to be described.

Reduction of threads requires exactly the same apparatus than reduction of any ML program, so we shall not detail these aspects longer. The functional sub-machine of the LCS abstract machine implements a customized and optimized SECD machine, it may be considered a cousin of Cardelli’s Functional Abstract Machine. We shall also omit to discuss the management of the store.

Handling process preemption

As noted in Section 4, some processes may be aborted upon actions performed by other processes. We shall say that those actions having pruning effects are *preemptive*, and that a process is *persistent* versus another if it resists a preemptive action of the latter. We shall also say that a thread is *obsolete* if it is not persistent with one of the threads that made a preemptive action since the beginning of evaluation. Considering that behaviors composed by choice or catch are here arbitrary behaviors, it would be costly in practice to take the pruning effects of preemptive actions into account as soon as the action is performed; LCS implementations rely on an original solution for implementing process preemption.

LCS abstract machines dynamically encode within the threads an information (the *position* information) from which the relative persistence of two threads can be efficiently determined. The consistency of the system of threads is enforced using a global register called the *position accumulator*. The accumulator holds an information dynamically computed from the position informations of all threads that made a preemptive action and permits to determine efficiently if some thread is obsolete or not. Every application of a reduction, expansion or communication rule to a thread should be preceded by a non-obsolescence test for that thread (in practice, only very few such tests are needed). Obsolete threads are removed when found; in addition, a separate mechanism incrementally collects and removes all obsolete threads from the system.

Local positions are determined from the inherited position and the composition operators. Seeing compound threads in Section 4 as trees, the position of a thread may be thought of as the path in this tree leading from the root to that thread. The global position accumulator, updated every time a process makes a preemptive action, may be thought of as the union of all paths to the threads that performed preemptive actions. We shall not discuss here concrete encoding for the position and accumulator registers.

Finally, note that two LCS threads may communicate if and only if one of them at least is persistent versus the other. So the information encoding persistence can also be used to determine if two particular threads may or not communicate.

The communication environment

Each thread maintains a structure that associates each possible communication port (constituted of a label and an extension) with a communication line. A line is a pair of possibly empty sets of suspensions: those offering input on that line, and those offering output on that line. Note that, due to the unrestricted choice composition, we may have both sets simultaneously nonempty while none of the threads in the input set may communicate with some in the output set. The lines are shared by all threads, while the communication environments are local.

Upon a restriction, a thread updates its communication environment so that ports with the restricted labels are all associated with “new” lines with empty suspension sets. A relabelling is handled by associating to the ports bearing the label being renamed the lines currently associated with the ports bearing the label in which it is renamed. Upon expansion of a composition operator, each thread creates inherits the current communication environment.

Upon an input (resp. output) offer, a thread searches a partner for communication in the output (resp. input) suspension set of the line associated with the port on which the offer is made. A partner is a non-obsolete thread which may communicate with it, this is determined from the positions of the two threads and the accumulator. If there is a possible partner, then the message is transmitted, the accumulator is updated and both threads are resumed. Otherwise, the thread having made the offer is suspended in the line (the relevant suspension set is physically updated, so that the new offer will be seen by all threads referencing that line).

Bounded parallelism machines, scheduling

Real machines obviously cannot provide an arbitrary number of processors for running all active threads. In practice, the set of active threads has to be partitioned and the threads in each subset in the partition share a single processor. Two implementations have been investigated: a sequentially running version [4], and a multi-processor version still in development [10] [11].

In the sequential case, where all threads share a single processor, a queue R of ready processes is added to the global registers, holding the active threads currently waiting for the processor. The suspension sets in the lines are handled as queues, too. Upon an unsatisfied communication offer, the next process in queue R is resumed; upon a communication, the receiver is resumed and the sender is suspended at the end of R . In addition to this “communication-bound” scheduling, a time-slicing mechanism prevents diverging or looping threads to take the processor forever. Since there is a single processor, the non-obsolescence test is only required when a thread is resumed.

In parallel implementations [11], the queue of ready processes is dynamically partitioned among the available processors with the help of a load-balancing algorithm. The process migration strategy also prevents the processes subject to preemption (i.e. those nonpersistent with some other process) to be moved from a processor to another so that all processors may work asynchronously, each with a private accumulator register.

6. Conclusion

Related work

LCS benefited from the results of the PFL experience, an early attempt to add CCS capabilities to ML [8]. PFL required a “continuation-passing” style of programming together with channels passed as parameters to the behaviors. LCS managed to keep the original CCS com-

binder set and concept of port, and more closely integrates behaviors with other features of the language.

CML [17] and the language described in [5] are based on different grounds than LCS; they provide a strictly functional interface for concurrency and communication. One of the difficulties with that approach is delaying communications while keeping an applicative order evaluation. In both these languages, communications are controlled by applying a synchronization function to the suspended communications. LCS uses a layered evaluation method for achieving transparently the same effects.

As LCS, the language FACILE [7] uses behavior expressions to describe processes, though restricted there to the termination behavior and compositions. However, the FACILE treatment of concurrency is closer to that of languages offering a functional interface.

Finally, the connections with the LOTOS effort by Brinksma et al. at the University of Twente [9] aimed at providing rigorous specification techniques for Open System Interconnection should be noted. Though both languages are based upon CCS for their behavioral aspects, LCS and LOTOS differ in nearly all other areas. LOTOS has a more specification-oriented flavor, due in part to the algebraic treatment of its non-behavioral features; LCS certainly has a more computational flavor.

Conclusion and further work

On one hand, LCS is a rigorously designed language including all capabilities of CCS and inheriting most of its theory. Parallel applications can be written abstractly enough so that users may reason about their programs. On another hand, the language has all the ingredients and facilities of a modern general purpose programming language. The language, including its behavioral primitives, was designed with efficiency of implementations as an important concern. The fact that abstract enough specifications and efficient enough implementations can be written in a single framework is certainly an advantage for formal development of programs.

LCS allows one to comfortably experiment with the parallel programming concepts introduced by CCS and related behavioral formalisms, and to gain experience in using these paradigms. It is also a valuable teaching tool for these concepts. Our experience with users of the system over the last years taught us that newcomers to parallel programming adapt quickly to the set of high level programming primitives provided by its behavioral interface; the behavioral approach helps intuition and, yet, is powerful enough to write real size applications.

Among the spin-offs of the experiment is an original method for typing behaviors that, in the more general setting of [3] can be used for typing a variety of other features of programming languages. Though it could not be described in depth here, the implementation of LCS also has a number of unique features such as both strong and light processes combined with automatic memory allocation, handling of unguarded choice compositions, etc. We hope to present the details of the implementations in forthcoming documents.

It is hoped that practising with the concepts implemented will prompt desirable refinements of the language, such as, for instance, introduction of some synchrony or stronger forms of label passing. Our short term goals include parallel implementations of the language; an architecture and some important building blocks have already been defined [10] [11].

Acknowledgments

Didier Giralt† and Jean-Paul Gouyon worked intensively on the project in its early stages. We are indebted to all those who helped implement the language or improve its design by their criticisms, including all users. We are especially grateful to Chris Reade for his comments and suggestions, and to Gérard Berry and colleagues at Sophia-Antipolis for a number of stimulating discussions. This work was partially supported by CNRS Greco C-Cube.

References

1. E. Artesiano, E. Zucca, "Parametric channels via label expressions in CCS*", Theoretical Computer Science, vol 33, 1984.
2. B. Berthomieu, "LCS, une implantation de CCS", 3ème Colloque C-Cube, A. Arnold Ed., Angoulême, France, September 1988.
3. B. Berthomieu, "Tagged Types – A theory of order sorted types for tagged expressions", LAAS Technical Report 93083, March 1993.
4. B. Berthomieu, D. Giralt, J.-P. Gouyon, "LCS user's manuals", LAAS Technical Report 91226, June 1991.
5. D. Berry, R. Milner, D. N. Turner, "A semantics for ML concurrency primitives", ACM Symposium on Principles of Programming Languages, 1992.
6. U. Engberg, M. Nielsen, "A Calculus of Communicating Systems with Label Passing", Research Report DAIMI PB-208, Computer Science Department, U. of Aarhus, 1986.
7. A. Giacalone, P. Mishra, S. Prasad, "Facile: A symmetric integration of concurrent and functional programming". Int. Journal of Parallel Programming, 18(2), April 1989.
8. S. Holstrom, "PFL: A Functional Language for Parallel Programming". In *Declarative Programming Workshop*, Chalmers U. of Technology, U. of Goteborg, Sweden, 1983.
9. ISO, "ISO-LOTOS, A Formal Description Technique based on the Temporal Ordering of Observational Behaviour", Int. Standard ISO 8807, ISO, 1989.
10. T. Le Sergent, B. Berthomieu, "Incremental multi-threaded garbage collection on virtually shared memory architectures", Int. Workshop on Memory Management, St. Malo, France, Sept. 1992.
11. T. Le Sergent, "Méthodes d'exécution et machines virtuelles parallèles pour l'implantation distribuée du langage de programmation parallèle LCS", Ph.D. thesis, Feb. 93.
12. D. Matthews, "A distributed concurrent implementation of Standard ML", EurOpen Autumn 1991 Conference, Budapest, Hungary, 1991.
13. R. Milner, *Communication and Concurrency*, Prentice Hall international series in Computer science, C.A.R. Hoare Ed, 1989.
14. R. Milner, M. Tofte, and R. Harper, *The Definition of Standard ML*, The MIT Press, 1990
15. R. Milner, J. Parrow, D. Walker, "A Calculus of Mobile Processes", ECS-LFCS-89-85, LFCS report series, Edinburgh University, 1989.
16. S. Owicky, D. Gries, "An Axiomatic Proof Technique for Parallel Programs I", Acta Informatica, vol 14, 1976.
17. J. H. Reppy, "CML: A higher-order concurrent language", ACM SIGPLAN Conf. on Programming Language Design and Implementation, SIGPLAN Notices 26(6), 1991.
18. D. Rémy, "Typechecking records and variants in a natural extension of ml", In Proceedings of the 16th ACM Symp. on Principles of Programming Languages, 1989.
19. B. Thomsen, "A calculus of higher order communicating systems", In Proceedings of the 16th ACM Symposium on Principles of Programming Languages, 1989.
20. M. Wand, "Complete Type Inference for Simple Objects", In Second Symposium on Logic in Computer Science, Ithaca, New York, june 1987.

Characterizing Behavioural Semantics and Abstractor Semantics

Michel Bidoit*, Rolf Hennicker**, Martin Wirsing**

*LIENS, CNRS & Ecole Normale Supérieure
45, Rue d'Ulm, 75230 Paris Cedex, FRANCE

**Institut für Informatik, Ludwig-Maximilians-Universität München
Leopoldstr. 11B, D-80802 München, GERMANY

Abstract. In the literature one can distinguish two main approaches to the definition of observational semantics of algebraic specifications. On one hand, observational semantics is defined using a notion of observational satisfaction for the axioms of a specification and on the other hand one can define observational semantics of a specification by abstraction with respect to an observational equivalence relation between algebras. In this paper we present an analysis and a comparative study of the different approaches in a more general framework which subsumes not only the observational case but also other examples like the bisimulation congruence of concurrent processes. Thereby the distinction between the different concepts of observational semantics is reflected by our notions of behavioural semantics and abstractor semantics. Our main results show that behavioural semantics can be characterized by an abstractor construction and, vice versa, abstractor semantics can be characterized in terms of behavioural semantics. Hence there exists a duality between both concepts which allows to express each one by each other. As a consequence we obtain a sufficient and necessary condition under which behavioural and abstractor semantics coincide. Moreover, the semantical characterizations lead to proof-theoretic results which show that behavioural theories of specifications can be reduced to standard theories (of some classes of algebras).

1 Introduction

Observability plays an important role in program development. For instance, formal implementation notions can be based on this concept. Other applications are the notion of equivalence between concurrent processes and the abstraction from single step transitions to input-output operational semantics.

Since the beginning of the eighties observational frameworks have found continuous interest in the area of algebraic specifications. In the literature one can distinguish two main possibilities for the definition of observational semantics. One is based on the so-called observational satisfaction relation where equations are not interpreted as identities but as observational equivalences of objects (cf. e.g. [Nivela, Orejas 88], [Bernot, Bidoit 91], [Hennicker 91]). In this case the observational semantics of a specification is given by the class of all algebras that observationally satisfy the axioms of the specification. Other approaches define observational semantics by constructing the closure of the (standard) model class of a specification with respect to an observational equivalence relation on algebras (cf. e.g. [Reichel 81], [Sannella, Tarlecki 85, 88], [Wirsing 86], [Schoett 87]). In [Reichel 85] both semantical views are considered and it is shown that they are equivalent if the axioms of a specification are conditional equations with observable premises. However, this is in general not true for specifications with arbitrary first-order formulas as axioms.

In this paper we study the relationships (and differences) between the two semantical concepts in a more general framework which allows to apply our results not only to the observational case but also to other examples like e.g. the bisimulation congruence of concurrent processes. Technically, we generalize the two views of observational semantics in the following way: Instead of the observational equivalence of elements

we use an arbitrary congruence relation for the interpretation of equations. This leads to our notion of *behavioural specification* which admits as models all algebras which satisfy the axioms of a specification with respect to a given congruence relation. On the other hand, following the notion of an "abstractor" in [Sannella, Tarlecki 88], we define *abstractor specifications* which describe all algebras that are equivalent to a (standard) model of a specification w.r.t. a given equivalence relation on algebras. In order to establish the connection between behavioural and abstractor semantics we consider only those equivalences on algebras which are "factorizable" (by a congruence relation between the elements of the algebras). An example of a factorizable equivalence is the observational equivalence of algebras w.r.t. a fixed set of observable sorts where the observable "experiments" can take arbitrary inputs. Equivalences which allow only observable inputs (cf. e.g. [Nivela, Orejas 88]) can be captured by generalizing our approach to partial congruences which, however, are not considered in this paper.

As a first central result of our approach we obtain that behavioural semantics can be characterized by the class of all algebras which are equivalent to a fully abstract (standard) model of the specification. This result implies, for instance, that behavioural semantics is more restrictive than abstractor semantics and that a behavioural specification is consistent if and only if there exists a fully abstract (standard) model of the specification. Conversely, we show that abstractor semantics can be characterized in terms of behavioural semantics as well. Hence there exists a nice duality between both kinds of semantics. Each one can be expressed by each other. As a consequence we obtain a necessary and sufficient condition for the equivalence of behavioural and abstractor specifications which says that behavioural semantics coincides with abstractor semantics if and only if the (standard) model class of a specification is closed under the "behavioural quotient" construction.

For the analysis of behavioural properties of specifications we consider their *behavioural theories*. According to the generalized satisfaction relation with respect to a congruence, the behavioural theory of a specification is defined as the set of all formulas which are satisfied w.r.t. the given congruence by all models of the specification. (In the observational framework this is exactly the set of all formulas which are observationally satisfied by all observational models of a specification.) Since it is usually difficult to prove behavioural theorems we need techniques which allow to reduce behavioural proofs to standard ones. Using our characterization of behavioural semantics we can show that the behavioural theory of a behavioural specification is the same as the standard theory of the class of the fully abstract (standard) models of the specification. Similarly we can use the characterization of abstractor semantics for showing that the behavioural theory of an abstractor specification can be reduced to the standard theory of the class of "behavioural quotients" of the (standard) models of the specification. These results provide the basis for the investigation of concepts which allow to prove behavioural properties of specifications by standard proof techniques (cf. [Bidoit, Hennicker 94]).

In this paper we consider flat specifications with first-order axioms. A generalization to structured specifications is possible but requires some more technical assumptions.

2 Algebraic Preliminaries

We assume the reader to be familiar with the basic notions of algebraic specifications (cf. e.g. [Ehrig, Mahr 85]) like *signature* $\Sigma = (S, F)$ (where S is a set of *sorts* and F is a set of *function symbols*), *total Σ -algebra* $A = ((A_S)_{S \in S}, (f^A)_{f \in F})$ (consisting of a

family of carrier sets $(A_s)_{s \in S}$ and a family of (total) functions $(f^A)_{f \in F}$, *term algebra* $T(\Sigma, X)$ over an S -sorted family X of variables of sort s .

If A is a total Σ -algebra then a *valuation* $\alpha: X \rightarrow A$ is a family of mappings $(\alpha_s: X_s \rightarrow A_s)_{s \in S}$. The *interpretation w.r.t.* α of a term $t \in T(\Sigma, X)$ is denoted by $I_\alpha(t)$. A Σ -congruence on A is a family of equivalence relations $\approx_A = (\approx_{A,S} \subseteq A_S \times A_S)_{S \in S}$ such that for all $f \in F$, f^A is compatible with \approx_A . The class of all Σ -algebras is denoted by $Alg(\Sigma)$.

3 Behavioural Specifications and Abstractor Specifications

In this section we will define the syntax and semantics of behavioural and abstractor specifications which both are built on top of standard specifications.

3.1 Standard Specifications

A standard specification is a flat, first-order specification where a distinguished set of function symbols is declared as constructors. The constructor declaration restricts the class of admissible models to those algebras which are finitely generated by the constructor symbols, i.e. all elements can be denoted by a constructor term (which is built only by constructor symbols and by variables of those sorts for which no constructor is defined).

Definition 3.1 Let $\Sigma = (S, F)$ be a signature, $Cons \subseteq F$ be a distinguished set of *constructors* and $X = (X_s)_{s \in S}$ a family of countably infinite sets X_s of variables of sort $s \in S$. Then a term t is called *constructor term* if $t \in T(\Sigma', X')$ where $\Sigma' = (S, Cons)$, $X' = (X_s)_{s \in S \setminus range(Cons)}$ and $range(Cons) = \{s \in S \mid \text{there exists } f \in Cons \text{ with functionality } s_1 \times \dots \times s_n \rightarrow s\}$. The set of constructor terms is denoted by T_{Cons} .

A Σ -algebra A is called *finitely generated by Cons* if for any $a \in A$ there exists a constructor term $t \in T_{Cons}$ and a valuation α such that $I_\alpha(t) = a$.

(Note that the definition of the generation principle is independent from the choice of X as long as X is countably infinite which is generally assumed here.) ♦

The axioms of a specification are Σ -formulas which are arbitrary first-order formulas over a multi-sorted signature Σ where the only predicate symbol is equality " $=$ ".

Definition 3.2 Let Σ be a signature. The set of (well-formed) Σ -formulas is inductively defined by:

- (0) If $t, r \in T(\Sigma, X)_s$ are terms of sort s , then $t = r$ is a Σ -formula (called equation).
- (1) If ϕ, ψ are Σ -formulas then $\neg\phi$, $\phi \wedge \psi$ and $\forall x:s.\phi$ are Σ -formulas.

All other logical operators (like \vee , \Rightarrow , \exists) are defined as usual. ♦

A Σ -algebra A satisfies a Σ -formula ϕ (denoted by $A \models \phi$) if A satisfies ϕ for all valuations $\alpha: X \rightarrow A$ (denoted by $A, \alpha \models \phi$) in the usual sense of first-order logic.

Definition 3.3

- (1) A *standard specification* $SP = (\Sigma, Cons, E)$ consists of a signature $\Sigma = (S, F)$, a distinguished subset $Cons \subseteq F$ of constructors and a set E of Σ -formulas, called *axioms of SP*.
- (2) The model class $Mod(SP)$ of SP is defined by $Mod(SP) =_{\text{def}} \{A \in Alg(\Sigma) \mid A \models \phi \text{ for all } \phi \in E \text{ and } A \text{ is finitely generated by } Cons\}$. ♦

Remark Note that if $\text{Cons} = \emptyset$ then any algebra $A \in \text{Alg}(\Sigma)$ is finitely generated by Cons . This means that in this case $\text{Mod}(\text{SP})$ is simply the class of all Σ -algebras satisfying the axioms of SP. Hence our assumption that any specification has a set of constructors is not a restriction but, on the contrary, it allows to apply our results also to specifications with reachability constraints. ♦

Example 3.4

1. The following specification SET is a usual specification of finite sets over arbitrary elements with constructors "true", "false" for the boolean values and "empty", "add" for sets. The operation "iselem" defines the membership test on sets.

```
spec SET =
  sorts {bool, elem, set}
  cons {true: → bool, false: → bool, empty: → set, add: elem, set → set}
  opns {iselem: elem, set → bool}
  axioms {∀ x, y: elem, s: set.
    iselem(x, empty) = false ∧ iselem(x, add(x, s)) = true ∧
    [x ≠ y ⇒ iselem(x, add(y, s)) = iselem(x, s)] ∧
    add(x, add(y, s)) = add(y, add(x, s)) ∧ add(x, add(x, s)) = add(x, s) }
endspec
```

For instance the algebra $P_{\text{fin}}(\mathbb{N})$ of finite subsets of the set \mathbb{N} of natural numbers is a model of SET.

2. The following specification CSO describes the operational semantics of a trivial nondeterministic sublanguage of CCS. It defines a sort "process" of processes containing a constant "nil", a semantical composition ".," of actions and processes and a nondeterministic choice operator "+". The operational semantics is given by a one-step transition function where $(p \xrightarrow{a} p') = \text{true}$ indicates that there is a transition from process p to process p' when executing the action a . All known equivalences on processes induce models of CSO.

```
spec CSO =
  sorts {bool, action, process}
  cons {true: → bool, false: → bool,
        nil : → process, . . . : action, process → process,
        . + . : process, process → process}
  opns {→ : process, action, process → bool}
  axioms {∀ a: action, p, p', q: process.
    (a, p  $\xrightarrow{a}$  p') = true ∧
    [(p  $\xrightarrow{a}$  p') = true ⇒ ((p+q  $\xrightarrow{a}$  p') = true ∧ (q+p  $\xrightarrow{a}$  p') = true)] }
endspec ♦
```

3.2 Behavioural Specifications

Behavioural specifications are a generalization of standard specifications which allow to describe the behaviour of data structures (or programs) with respect to a given congruence relation. For this purpose we first generalize the standard satisfaction relation. The only difference between this generalization and the standard satisfaction relation of the first-order predicate calculus is that the predicate symbol $=$ is not interpreted by the set-theoretic equality over the carrier sets of an algebra but by a given congruence relation (a connection between both satisfaction relations will be established in Proposition 3.10.1).

Definition 3.5 Let A be a Σ -algebra and \approx_A be a Σ -congruence on A . Moreover, let $t, r \in T(\Sigma, X)_S$ be terms of sort s . The *satisfaction relation w.r.t. \approx_A* (denoted by $\models_{=A}$) is defined as follows:

- (0) For any valuation $\alpha: X \rightarrow A$: $A, \alpha \models_{=A} t = r$ if $I_\alpha(t) \approx_A I_\alpha(r)$.
- (1) The satisfaction relation w.r.t. \approx_A for arbitrary Σ -formulas (cf. Definition 3.2) is defined (as usual) by induction over the structure of the formulas.
- (2) For any Σ -formula ϕ : $A \models_{=A} \phi$ if $A, \alpha \models_{=A} \phi$ for all valuations $\alpha: X \rightarrow A$. ♦

In a similar way we also generalize the generation principle of algebras (by a set of constructors) with respect to a congruence:

Definition 3.6 Let $\Sigma = (S, F)$ be a signature, $Cons \subseteq F$ a distinguished set of constructors, A a Σ -algebra and \approx_A a Σ -congruence on A . A is called \approx_A -finitely generated by $Cons$ if for any $a \in A$ there exists a constructor term $t \in T_{Cons}$ and a valuation α such that $I_\alpha(t) \approx_A a$. In particular, if $Cons = \emptyset$ then any algebra $A \in Alg(\Sigma)$ is \approx_A -finitely generated by $Cons$. (Proposition 3.10.2 provides a connection between the standard generation principle and the generation principle w.r.t. \approx). ♦

Example 3.7

1. Observational equivalence: An important example of a congruence is the observational equivalence of objects which is used for the interpretation of equations in several observational approaches in the literature (cf. above). Formally, we assume given a signature $\Sigma = (S, F)$ and a distinguished set $Obs \subseteq S$ of observable sorts (which denote the carrier sets of observable values). Then two objects of a Σ -algebra A are considered to be observationally equivalent, if they cannot be distinguished by "experiments" with observable result. This can be formally expressed using the notion of *observable context*, which is any term $c \in T(\Sigma, X \cup Z)$ of observable sort which contains (besides variables in X) exactly one variable $z_S \in Z$. (Thereby $Z = \{z_S \mid s \in S\}$ is an S -sorted set of variables such that $z_S \notin X_s$ for all $s \in S$.) The observational equivalence of objects can now be defined in the following way:

For any $s \in S$, two elements $a, b \in A_s$ are called *observationally equivalent* (denoted by $a \approx_{Obs, A} b$) if for all observable Σ -contexts c containing z_S and for all valuations $\alpha: X \rightarrow A$, $I_{\alpha 1}(c) = I_{\alpha 2}(c)$ holds where $\alpha 1(z_S) = a$, $\alpha 2(z_S) = b$ and $\alpha 1(x) = \alpha 2(x) = \alpha(x)$ for all $x \in X$.

It is easy to show that $\approx_{Obs, A}$ defines a congruence relation on A . The satisfaction relation w.r.t. $\approx_{Obs, A}$ is often called *observational* (or *behavioural*) *satisfaction relation*. (Note that for observable equations $t = r$ where t and r are of observable sort the observational satisfaction relation coincides with the standard one, i.e. $A \models_{=Obs, A} t = r$ if and only if $A \models t = r$.)

2. Strong bisimulation: The notion of strong bisimulation is an example of a congruence on CSO. For any algebra A over the signature of CSO we define the notion of *simulation equivalence* as follows: Let $p, q \in A_{process}$. Then $p \approx_{sim, A} q$ if for all $p' \in A_{process}$, $s \in (A_{action})^*$,

$[(p \xrightarrow{s} p') = true^A \Leftrightarrow \exists q' \in A_{process}, p' \approx_{sim, A} q' \text{ and } (q \xrightarrow{s} q') = true^A]$ and vice versa for all $q' \in A_{process}$, $s \in (A_{action})^*$,

$[(q \xrightarrow{s} q') = true^A \Leftrightarrow \exists p' \in A_{process}, q' \approx_{sim, A} p' \text{ and } (p \xrightarrow{s} p') = true^A]$.

Thereby \rightarrow^* denotes the reflexive and transitive closure of \rightarrow .

According to [Astesiano, Wirsing 89] we have the following fact: Let A be a (standard) model of CSO. Then any simulation equivalence is a congruence (w.r.t. the signature of CSO) and the simulation congruences on A form a complete lattice. For

any algebra A , the coarsest simulation congruence on A is called strong bisimulation on A and is denoted by $\approx_{\text{bisim}, A}$. If for A we choose the Herbrand model $H(\text{CSO})$ of CSO with the set T_{Cons} of constructor terms as carrier sets and the following interpretation of " \rightarrow " by provable transitions

$H(\text{CSO}) \Vdash (p \xrightarrow{a} q) = \text{true}$ iff $\text{CSO} \Vdash (p \xrightarrow{a} q) = \text{true}$,
then $\approx_{\text{bisim}, H(\text{CSO})}$ is Milner's strong bisimulation congruence.

As an example for $\approx_{\text{bisim}, H(\text{CSO})}$ consider the processes

$p_1 =_{\text{def}} a.(c.\text{nil} + d.\text{nil})$, $p_2 =_{\text{def}} a.c.\text{nil} + a.d.\text{nil}$, and $p_3 =_{\text{def}} a.d.\text{nil} + a.c.\text{nil}$ (with pairwise different a, c, d). Then $H(\text{CSO}) \Vdash \neg(p_1 = p_2) \wedge p_2 = p_3$, but for the standard satisfaction we have obviously $H(\text{CSO}) \Vdash \neg(p_1 = p_2) \wedge \neg(p_2 = p_3)$. ♦

Behavioural specifications can be built on top of standard specifications for any given family $\approx = (\approx_A)_{A \in \text{Alg}(\Sigma)}$ of Σ -congruences on the algebras $A \in \text{Alg}(\Sigma)$ (cf. also the notion of observational Σ -algebra in [Knapik 91]). The essential difference to standard specifications is that instead of the standard satisfaction relation the satisfaction relation w.r.t. $=$ is used for the interpretation of the axioms.

Definition 3.8 Let $\text{SP} = (\Sigma, \text{Cons}, E)$ be a standard specification and let $\approx = (\approx_A)_{A \in \text{Alg}(\Sigma)}$ be a family of Σ -congruences. Then:

- (1) **behaviour SP wrt $=$** is a *behavioural specification*.
- (2) The model class of a behavioural specification consists of all Σ -algebras A which satisfy the axioms of SP w.r.t. $=_A$ and which are $=_A$ -finitely generated by the constructors Cons , i.e.

$$\text{Mod}(\text{behaviour SP wrt } =) =_{\text{def}} \{A \in \text{Alg}(\Sigma) \mid A \Vdash =_A \phi \text{ for all } \phi \in E \text{ and } A \text{ is } =_A\text{-finitely generated by Cons}\}. \quad \diamond$$

Example 3.9

1. Let $\Sigma = (S, F)$ be a signature and $\text{Obs} \subseteq S$ be a set of observable sorts. The set Obs induces a family $\approx_{\text{Obs}} = (\approx_{\text{Obs}, A})_{A \in \text{Alg}(\Sigma)}$ of Σ -congruences where for any $A \in \text{Alg}(\Sigma)$, $\approx_{\text{Obs}, A}$ is the observational equivalence defined in Example 3.7.1. Then a specification **behaviour SP wrt \approx_{Obs}** specifies the *observable behaviour* of a data structure (or a program) by means of the observational satisfaction relation.

As a concrete example we can construct the behavioural specification

behaviour SET wrt $=_{\{\text{bool}, \text{elem}\}}$

on top of the standard specification **SET** of sets (cf. Example 3.4.1). Here $\text{Obs} = \{\text{bool}, \text{elem}\}$, i.e. the sorts "bool" and "elem" are considered as observable. Since the sort "set" is not observable, sets can only be observed via the membership test "iselem". For instance, the algebra N^* of finite sequences of natural numbers is a model of the behavioural specification of sets. In particular N^* satisfies observationally the last two axioms of **SET**, because one cannot distinguish the order of the elements and the number of occurrences of elements in a sequence by the allowed observations. But note that N^* does not satisfy the last two **SET** axioms w.r.t. the standard satisfaction relation and hence is not a model of the standard specification **SET**.

2. The behavioural specification **behaviour CSO wrt \approx_{bisim}** describes all algebras (over the signature of **CSO**) that satisfy the axioms of **CSO** w.r.t. the strong bisimulation congruence. But since **CSO** does not require any equations between processes we have $\text{Mod}(\text{CSO}) = \text{Mod}(\text{behaviour CSO wrt } \approx_{\text{bisim}})$. ♦

The following proposition establishes an important connection between the generalized satisfaction w.r.t. \approx and the standard satisfaction of formulas and analogously between both kinds of generation principles:

Proposition 3.10 Let $\Sigma = (S, F)$ be a signature, $= = (\approx_A)_{A \in \text{Alg}(\Sigma)}$ be a family of Σ -congruences and $\text{Cons} \subseteq F$ be a set of constructors. Then for all Σ -algebras A the following holds:

- (1) For all Σ -formulas ϕ , $A \models_{=A} \phi$ if and only if $A/\approx_A \models \phi$,
(A/\approx_A denotes the quotient algebra of A w.r.t. \approx_A)
- (2) A is $=_A$ -finitely generated by Cons if and only if A/\approx_A is finitely generated by Cons .

Sketch of the proof: (1) The proof follows from the following lemma which can be proved by induction on the form of ϕ : For any valuation $\alpha: X \rightarrow A: A$, $\alpha \models_{=A} \phi$ if and only if $A/\approx_A, \pi \circ \alpha \models \phi$ where $\pi: A \rightarrow A/\approx_A$ is the canonical epimorphism.

(2) The proof is straightforward. ♦

As a first consequence of Proposition 3.10 we obtain:

Corollary 3.11 For any Σ -algebra A the following holds:

$$A \in \text{Mod}(\text{behaviour SP wrt } =) \text{ if and only if } A/\approx_A \in \text{Mod}(\text{SP}).$$

3.3 Abstractor Specifications

The notion of "abstractor" was introduced in [Sannella, Tarlecki 88] for describing a specification building operation which allows to abstract from the model class of a specification with respect to a given equivalence relation on the class of all Σ -algebras.

Definition 3.12 Let $\equiv \subseteq \text{Alg}(\Sigma) \times \text{Alg}(\Sigma)$ be an equivalence relation on $\text{Alg}(\Sigma)$. For any class $C \subseteq \text{Alg}(\Sigma)$ of Σ -algebras, $\text{Abs}_{\equiv}(C)$ denotes the closure of C under \equiv , i.e.

$$\text{Abs}_{\equiv}(C) =_{\text{def}} \{B \in \text{Alg}(\Sigma) \mid B \equiv A \text{ for some } A \in C\}. \quad \diamond$$

The syntax and semantics of abstractor specifications is defined by:

Definition 3.13 Let $\text{SP} = (\Sigma, \text{Cons}, E)$ be a standard specification and let $\equiv \subseteq \text{Alg}(\Sigma) \times \text{Alg}(\Sigma)$ be an equivalence relation. Then:

- (1) **abstract SP wrt \equiv** is an *abstractor specification*.
- (2) The model class of an abstractor specification is the closure of $\text{Mod}(\text{SP})$ under \equiv :
 $\text{Mod}(\text{abstract SP wrt } \equiv) =_{\text{def}} \text{Abs}_{\equiv}(\text{Mod}(\text{SP})). \quad \diamond$

3.4 Relating Congruences of Elements and Equivalences of Algebras

In this paper we are interested in a relationship between behavioural specifications and abstractor specifications. For this purpose we have to find a connection between congruences between elements of algebras (which are used to define behavioural specifications) and equivalences between algebras themselves (which are used to define abstractor specifications). If we start from a family of Σ -congruences then it is an easy task to construct an associated equivalence relation between algebras in the following way:

Definition 3.14 Let $= = (=_A)_{A \in \text{Alg}(\Sigma)}$ be a family of Σ -congruences. Then $\equiv_{=}$ $\subseteq \text{Alg}(\Sigma) \times \text{Alg}(\Sigma)$ is the following equivalence relation on $\text{Alg}(\Sigma)$:

For any $A, B \in \text{Alg}(\Sigma)$, $A \equiv_{=} B$ if $A/_= A$ and $B/_= B$ are isomorphic. ♦

If we start from an equivalence relation on $\text{Alg}(\Sigma)$ we find an associated family of Σ -congruences only if the equivalence is "factorizable":

Definition 3.15 An equivalence relation $\equiv \subseteq \text{Alg}(\Sigma) \times \text{Alg}(\Sigma)$ is called *factorizable* if there exists a family $= = (=_A)_{A \in \text{Alg}(\Sigma)}$ of Σ -congruences such that for all $A, B \in \text{Alg}(\Sigma)$ the following holds:

$A \equiv B$ if and only if $A/_= A$ and $B/_= B$ are isomorphic.

In this case we say that \equiv is *factorizable by* \approx . ♦

It is usually not a simple task to prove that an equivalence is factorizable. The following example shows how this can be done for the W -equivalence of ASL (cf. [Wirsing 86]) if W is the set of all observable terms with arbitrary input variables. For dealing with observable ground terms or with observable terms that allow only observable inputs our approach can be generalized by considering "partial" congruences (cf. the remarks in Section 6).

Example 3.16 Let $\Sigma = (S, F)$ be a signature and $\text{Obs} \subseteq S$ be a set of observable sorts. According to the W -equivalence relation of ASL we say that two Σ -algebras A and B are W -equivalent, denoted by $A \equiv_W B$, if there exists a family $Y = (Y_s)_{s \in S}$ of sets Y_s of variables of sort s and two surjective valuations $\alpha 1: Y \rightarrow A$ and $\beta 1: Y \rightarrow B$ such that for all terms $t, r \in T(\Sigma, Y)_s$ of observable sort $s \in \text{Obs}$ the following holds:

$$I_{\alpha 1}(t) = I_{\alpha 1}(r) \quad \text{if and only if} \quad I_{\beta 1}(t) = I_{\beta 1}(r).$$

We will now show that \equiv_W is factorizable by the family \approx_{Obs} of congruences defined in Example 3.9.1. For this purpose we have to prove that $A \equiv_W B$ holds if and only if $A/\approx_{\text{Obs}, A}$ and $B/\approx_{\text{Obs}, B}$ are isomorphic.

Let us first assume $A \equiv_W B$. Then using the definition of \equiv_W one can prove that for all terms $t, r \in T(\Sigma, Y)$ the following holds: $I_{\alpha 1}(t) \approx_{\text{Obs}, A} I_{\alpha 1}(r)$ if and only if $I_{\beta 1}(t) \approx_{\text{Obs}, B} I_{\beta 1}(r)$. Using this fact one can show that $h: A/\approx_{\text{Obs}, A} \rightarrow B/\approx_{\text{Obs}, B}$, $h([a]) =_{\text{def}} [I_{\beta 1}(t)]$ if $t \in T(\Sigma, Y)$ with $[a] = [I_{\alpha 1}(t)]$ defines a Σ -isomorphism between the quotient algebras.

Conversely let $h: A/\approx_{\text{Obs}, A} \rightarrow B/\approx_{\text{Obs}, B}$ be a Σ -isomorphism. W.l.o.g we assume that the carrier sets of A and B are disjoint. Then let $Y =_{\text{def}} A \cup B$, let $\alpha 1: Y \rightarrow A$ be defined by $\alpha 1(y) =_{\text{def}} y$ if $y \in A$, $\alpha 1(y) =_{\text{def}} a$ if $y \in B$ and $h([a]) = [y]$ and let $\beta 1: Y \rightarrow B$ be defined by $\beta 1(y) =_{\text{def}} y$ if $y \in B$, $\beta 1(y) =_{\text{def}} b$ if $y \in A$ and $h([y]) = [b]$. Using the definition of Y , $\alpha 1$ and $\beta 1$ we can prove by structural induction that for all terms $t \in T(\Sigma, Y)$, $h([I_{\alpha 1}(t)]) = [I_{\beta 1}(t)]$ holds. Now assume $I_{\alpha 1}(t) = I_{\alpha 1}(r)$ for two observable terms $t, r \in T(\Sigma, Y)_s$ with $s \in \text{Obs}$. Then $[I_{\alpha 1}(t)] = [I_{\alpha 1}(r)]$. Since $h([I_{\alpha 1}(t)]) = [I_{\beta 1}(t)]$ and $h([I_{\alpha 1}(r)]) = [I_{\beta 1}(r)]$ we obtain $[I_{\beta 1}(t)] = [I_{\beta 1}(r)]$, i.e. $I_{\beta 1}(t) \approx_{\text{Obs}, B} I_{\beta 1}(r)$. Then $I_{\beta 1}(t) = I_{\beta 1}(r)$ holds since t and r are of observable sort. Symmetrically one shows that for any observable terms t and r , $I_{\beta 1}(t) = I_{\beta 1}(r)$ implies $I_{\alpha 1}(t) = I_{\alpha 1}(r)$. Hence A and B are W -equivalent.

As a consequence we obtain that if Obs is a set of observable sorts then any abstractor specification **abstract SP wrt** \equiv_W defines an *observational abstraction* which has the same models as the specification **abstract SP wrt** \equiv_{Obs} where \equiv_{Obs} is the equivalence generated by the congruence \approx_{Obs} . ♦

Behavioural specifications (as introduced in Section 3.2) and abstractor specifications are based on the same intention, namely to allow a more general view of the semantics of specifications. In particular this is useful for formal implementation definitions where implementations may relax (some of) the properties of a given requirement specification (cf. e.g. abstractor implementations in [Sannella, Tarlecki 88] or behavioural implementations in [Hennicker 91], for a survey on implementation concepts and observability see [Orejas et al. 91]). However, the semantical definitions of behavioural specifications and abstractor specifications are quite different. Therefore it is an important issue to compare both approaches carefully and to figure out precisely the relationships and the differences between the two concepts. This is the topic of the next sections.

4 Characterization of Behavioural and Abstractor Semantics

If we consider the particular case of observable behaviour specifications (cf. Example 3.9.1) and observational abstractions (cf. Example 3.16) then we can conclude from a result in [Reichel 85] that both specifications have the same semantics if the axioms of the specification are conditional equations with observable premises. However, in the observational framework this is in general not true if the axioms are arbitrary first-order formulas. For instance, the following specification DEMO has a standard model which, by definition, is also a model of the abstractor specification "**abstract DEMO wrt** \equiv_{Obs} " but which is not a model of the behavioural specification "**behaviour DEMO wrt** \approx_{Obs} " if $\text{Obs} = \{s\}$, i.e. if s is the only observable sort.

```
spec DEMO =
  sorts { s, s' }
  opns { a, b: → s, c, d : → s', f: s' → s }
  axioms { f(c) = f(d) ∧ [c = d ⇒ a = b] }
endspec
```

Any Σ -algebra A where the constants a, b, c, d are interpreted as pairwise different objects and where f takes the same value for c and d is a standard model of the specification and hence also a model of "**abstract DEMO wrt** \equiv_{Obs} ". But such an algebra A is not a model of "**behaviour DEMO wrt** \approx_{Obs} " because it does not observationally satisfy (cf. Example 3.7.1) the second axiom: Since the only observable context for c and d is $f(z')$, the equation $c = d$ is observationally satisfied by A but $a = b$ is not observationally satisfied by A since a and b are different objects of observable sort.

Let us now come back to the more general situation. According to the relationship between a family $\approx = (\approx_A)_{A \in \text{Alg}(\Sigma)}$ of Σ -congruences and a factorizable equivalence relation \equiv established in Section 3.4, we can disregard whether we start from one point of view or from the other one. Hence, in the sequel we assume given a signature Σ , a set Cons of constructors and a couple (\approx, \equiv) consisting of a family $\approx = (\approx_A)_{A \in \text{Alg}(\Sigma)}$ of Σ -congruences and an equivalence relation \equiv between Σ -algebras such that \equiv is factorizable by \approx .

In order to obtain our central theorems we need some compatibility properties:

4.1 Compatibility Properties

The following proposition shows that an equivalence \equiv which is factorizable by a congruence \approx is compatible with the satisfaction relation w.r.t. \approx and with the generation principle w.r.t. \approx .

Proposition 4.1 If $A \equiv B$ are equivalent Σ -algebras, then the following holds:

- (1) For all Σ -formulas ϕ , $A \models_{=A} \phi$ if and only if $B \models_{=B} \phi$,
- (2) A is \approx_A -finitely generated by Cons if and only if B is \approx_B -finitely generated by Cons.

Proof: (1) Since $A \equiv B$ are equivalent and since \equiv is factorizable by \approx , A/\approx_A and B/\approx_B are isomorphic. By Proposition 3.10.1 we know that $A \models_{=A} \phi$ iff $A/\approx_A \models \phi$. Since isomorphic algebras satisfy the same Σ -formulas we have $A/\approx_A \models \phi$ iff $B/\approx_B \models \phi$ and again by Proposition 3.10.1 we obtain $B/\approx_B \models \phi$ iff $B \models_{=B} \phi$.

(2) The proof is done analogously to (1) using Proposition 3.10.2. ♦

4.2 Fully Abstract Algebras

An important role for the characterization of behavioural and abstractor semantics is played by fully abstract algebras. Following Milner's notion (cf. [Milner 77]) we define full abstractness with respect to a given family \approx of Σ -congruences in the following way:

Definition 4.2

- (1) A Σ -algebra A is called *fully abstract with respect to \approx* (or briefly \approx -fully abstract) if \approx_A is the equality on A , i.e. $\approx_A =_A$.
- (2) For any class $C \subseteq \text{Alg}(\Sigma)$ of Σ -algebras, $\text{FA}_{\approx}(C)$ denotes the subclass of \approx -fully abstract algebras of C , i.e. $\text{FA}_{\approx}(C) =_{\text{def}} \{A \in C \mid A \text{ is } \approx\text{-fully abstract}\}$. ♦

Example 4.3 If we consider the observational framework then elements of fully abstract algebras w.r.t. \approx_{Obs} are equal if and only if they are observationally equivalent. For instance, in the SET example the algebra $P_{\text{fin}}(N)$ is fully abstract while N^* is not. ♦

Lemma 4.4 If $A \in \text{Alg}(\Sigma)$ is \approx -fully abstract, then the following holds:

- (1) For all Σ -formulas ϕ , $A \models_{=A} \phi$ if and only if $A \models \phi$,
- (2) A is \approx_A -finitely generated by Cons if and only if A is finitely generated by Cons.

Proof: Since A is \approx -fully abstract, the congruence \approx_A is the equality $=_A$ on A and therefore (1) and (2) are trivially satisfied. ♦

Definition 4.5 A family $\approx = (\approx_A)_{A \in \text{Alg}(\Sigma)}$ of Σ -congruences is called *regular* if for any $A \in \text{Alg}(\Sigma)$ the quotient algebra A/\approx_A is \approx -fully abstract (i.e. $\approx(A/\approx_A) = =_{(A/\approx_A)}$). ♦

Example 4.6 Any family \approx_{Obs} which is generated by a set Obs of observable sorts (cf. Example 3.9.1) is regular. (In fact it seems that all reasonable examples of families of Σ -congruences are regular.) ♦

Lemma 4.7 If \approx is regular, then for any $A \in \text{Alg}(\Sigma)$, $A \equiv A/\approx_A$.

Proof: Since \approx is regular, \approx_{A/\approx_A} is the equality $=_{A/\approx_A}$. Hence the algebras A/\approx_A and $(A/\approx_A)/\approx_{A/\approx_A}$ are isomorphic. Since \equiv is factorizable by \approx this means that $A \equiv A/\approx_A$ holds. ♦

4.3 Characterization Theorems

We are now prepared to prove our first central theorem which says that for any regular congruence \approx the model class of a behavioural specification "behaviour SP wrt \approx " coincides with the closure of the class of fully abstract models of the standard specification SP under any equivalence relation which is factorizable by \approx . In the sequel of this paper we assume that \approx is regular.

Theorem 4.8 Let $SP = (\Sigma, \text{Cons}, E)$ be a standard specification. Then:

$$\text{Mod}(\text{behaviour SP wrt } \approx) = \text{Abs}_{\equiv}(\text{FA}_{\approx}(\text{Mod}(SP))).$$

Proof: \subseteq : Let $A \in \text{Mod}(\text{behaviour SP wrt } \approx)$. Then, by Corollary 3.11, $A/\approx_A \in \text{Mod}(SP)$. Since \approx is regular, A/\approx_A is \approx -fully abstract and, by Lemma 4.7, $A \equiv A/\approx_A$ are equivalent. Hence $A \in \text{Abs}_{\equiv}(\text{FA}_{\approx}(\text{Mod}(SP)))$.

\supseteq : Let $A \in \text{Abs}_{\equiv}(\text{FA}_{\approx}(\text{Mod}(SP)))$. Then $A \equiv B$ for some $B \in \text{FA}_{\approx}(\text{Mod}(SP))$. For proving that $A \in \text{Mod}(\text{behaviour SP wrt } \approx)$ we have to show that $A \models_{=A} \phi$ for any axiom $\phi \in E$ and that A is \approx_A -finitely generated by Cons. Now let $\phi \in E$ be an arbitrary axiom. Since B is a model of SP , $B \models \phi$ and since B is \approx -fully abstract, $B \models_{=B} \phi$ (by Lemma 4.4.1). Then, since $A \equiv B$, Proposition 4.1.1 shows that $A \models_{=A} \phi$. Analogously one can prove that A is \approx_A -finitely generated by Cons, using Lemma 4.4.2 and Proposition 4.1.2. ♦

The characterization of Theorem 4.8 gives rise to the definition of a semantical operator, called Beh_{\approx} , which can be applied to any class C of Σ -algebras and which corresponds on the semantical level to the syntactic behaviour operator (in the same way as the semantic Abs_{\equiv} operator corresponds to the syntactic abstract operator):

Definition 4.9 For any class $C \subseteq \text{Alg}(\Sigma)$ of Σ -algebras,

$$\text{Beh}_{\approx}(C) =_{\text{def}} \text{Abs}_{\equiv}(\text{FA}_{\approx}(C)). \quad \diamond$$

Using this definition we obtain: $\text{Mod}(\text{behaviour SP wrt } \approx) = \text{Beh}_{\approx}(\text{Mod}(SP))$ for any standard specification SP .

In Theorem 4.8 the semantics of behavioural specifications is characterized in terms of the semantical abstractor operator Abs_{\equiv} . In the following we will show that vice versa the semantics of abstractor specifications can be characterized using the semantical behaviour operator Beh_{\approx} . Hence there exists a duality between both kinds of semantics. Each one can be expressed by each other.

For the characterization of abstractor semantics given in Theorem 4.11 we use the following quotient construction:

Definition 4.10 For any class $C \subseteq \text{Alg}(\Sigma)$ of Σ -algebras, C/\approx denotes the class of all \approx -quotients of algebras of C , i.e. $C/\approx =_{\text{def}} \{A/\approx_A \mid A \in C\}$. ♦

Theorem 4.11 Let $SP = (\Sigma, \text{Cons}, E)$ be a standard specification. Then:

$$\text{Mod}(\text{abstract } SP \text{ wrt } \equiv) = \text{Beh}_{\equiv}(\text{Mod}(SP)/\approx).$$

Proof: By definition, $\text{Mod}(\text{abstract } SP \text{ wrt } \equiv) = \text{Abs}_{\equiv}(\text{Mod}(SP))$. Since \approx is regular, Lemma 4.7 shows that for any $A \in \text{Mod}(SP)$, $A \equiv A/\approx_A$. Hence $\text{Abs}_{\equiv}(\text{Mod}(SP)) = \text{Abs}_{\equiv}(\text{Mod}(SP)/\approx)$. Moreover, A/\approx_A is \approx -fully abstract and therefore $\text{Mod}(SP)/\approx = \text{FA}_{\approx}(\text{Mod}(SP)/\approx)$. In summary we have $\text{Mod}(\text{abstract } SP \text{ wrt } \approx) = \text{Abs}_{\approx}(\text{Mod}(SP)) = \text{Abs}_{\approx}(\text{Mod}(SP)/\approx) = \text{Abs}_{\approx}(\text{FA}_{\approx}(\text{Mod}(SP)/\approx)) = \text{Beh}_{\approx}(\text{Mod}(SP)/\approx)$. ♦

Remark 4.12 Since Beh_{\approx} is defined for any class C of Σ -algebras one can extend the construction of behavioural specifications to arbitrary structured specifications SP of some ASL-like specification language by defining $\text{Mod}(\text{behaviour } SP \text{ wrt } \approx) =_{\text{def}} \text{Beh}_{\approx}(\text{Mod}(SP))$. (Obviously, "abstract" can be extended as well.) Then, if we assume that the specification language contains the ASL-operator $.+$ for the combination of specifications and a quotient operator $/\approx$ with $\text{Mod}(SP/\approx) =_{\text{def}} \text{Mod}(SP)/\approx$ (and if we assume that any algebra A is identified with its trivial quotient A/A) we can prove using the above theorems the following equations which show that behavioural specifications can be expressed by abstractor specifications and vice versa:

$$\begin{aligned} \text{Mod}(\text{behaviour } SP \text{ wrt } \approx) &= \text{Mod}(\text{abstract } (SP/\approx + SP) \text{ wrt } \equiv) \text{ and} \\ \text{Mod}(\text{abstract } SP \text{ wrt } \equiv) &= \text{Mod}(\text{behaviour } (SP/\approx) \text{ wrt } \approx). \end{aligned} \quad \diamond$$

4.4 Consequences of the Characterization Theorems

Theorem 4.8 has various consequences. It shows that behavioural semantics is a subclass of abstractor semantics, that fully abstract models of a standard specification are behavioural models and that a behavioural specification is consistent if and only if there exists a fully abstract model of the underlying standard specification. The following corollary states also corresponding properties of abstractor specifications which immediately follow from the definition of abstractor semantics.

Corollary 4.13 Let $SP = (S, \text{Cons}, E)$ be a standard specification. Then:

- (1) $\text{Mod}(\text{behaviour } SP \text{ wrt } \approx) \subseteq \text{Mod}(\text{abstract } SP \text{ wrt } \equiv)$,
- (2) $\text{FA}_{\approx}(\text{Mod}(SP)) \subseteq \text{Mod}(\text{behaviour } SP \text{ wrt } \approx)$,
- (3) $\text{Mod}(\text{behaviour } SP \text{ wrt } \approx) \neq \emptyset$ if and only if $\text{FA}_{\approx}(\text{Mod}(SP)) \neq \emptyset$,
- (4) $\text{Mod}(SP) \subseteq \text{Mod}(\text{abstract } SP \text{ wrt } \equiv)$,
- (5) $\text{Mod}(\text{abstract } SP \text{ wrt } \equiv) \neq \emptyset$ if and only if $\text{Mod}(SP) \neq \emptyset$.

As a further important consequence of the characterization theorems we obtain the following necessary and sufficient conditions under which behavioural semantics and abstractor semantics coincide. Note that a particular application of condition (3) leads to the theorem of [Reichel 85] since in the case of conditional equational axioms with observable premises the model class of a standard specification is closed under the observational quotient construction.

Corollary 4.14 Let $SP = (\Sigma, \text{Cons}, E)$ be a standard specification. The following conditions are equivalent:

- (1) $\text{Mod}(\text{behaviour } SP \text{ wrt } \approx) = \text{Mod}(\text{abstract } SP \text{ wrt } \equiv)$,
- (2) $\text{Mod}(SP) \subseteq \text{Mod}(\text{behaviour } SP \text{ wrt } \approx)$,
- (3) $\text{Mod}(SP)/\approx \subseteq \text{Mod}(SP)$.

Proof: (1) \Rightarrow (2): Since $\text{Mod}(\text{SP}) \subseteq \text{Mod}(\text{abstract SP wrt } \equiv)$ (2) follows immediately from (1).

(2) \Rightarrow (3): We have to show that for any $A \in \text{Mod}(\text{SP})$, $A/\approx_A \in \text{Mod}(\text{SP})$ holds. Let $A \in \text{Mod}(\text{SP})$ be an arbitrary model. (2) implies that $A \in \text{Mod}(\text{behaviour SP wrt } \approx)$. Then, by Corollary 3.11, we obtain $A/\approx_A \in \text{Mod}(\text{SP})$.

(3) \Rightarrow (1): By Corollary 4.14.1.

\square : By Theorem 4.11, $\text{Mod}(\text{abstract SP wrt } \equiv) = \text{Beh}_=(\text{Mod}(\text{SP})/\approx)$. Now (3) implies $\text{Beh}_=(\text{Mod}(\text{SP})/\approx) \subseteq \text{Beh}_=(\text{Mod}(\text{SP}))$ and, by Theorem 4.8 together with the definition of $\text{Beh}_=$, we obtain $\text{Beh}_=(\text{Mod}(\text{SP})) = \text{Mod}(\text{behaviour SP wrt } \approx)$. \diamond

5 Theories of Behavioural and Abstractor Specifications

According to the generalization of the standard satisfaction relation to the satisfaction relation with respect to a congruence $=$ we will consider here for any behavioural or abstractor specification the theory with respect to $=$, i.e. the set of all Σ -formulas which are satisfied w.r.t. $=$ by all models of a behavioural or abstractor specification. We recall that we assume given also in this section a pair (\approx, \equiv) consisting of a regular family \approx of Σ -congruences and an equivalence relation \equiv on $\text{Alg}(\Sigma)$ which is factorizable by \approx .

Definition 5.1 For any class $C \subseteq \text{Alg}(\Sigma)$ of Σ -algebras, $\text{Th}_=(C)$ denotes the set of all Σ -formulas ϕ which are satisfied w.r.t. $=$ by all algebras of C , i.e.

$$\text{Th}_=(C) = \text{def } \{\phi \mid A \models_A \phi \text{ for all } A \in C\}.$$

$\text{Th}_=(C)$ is called \approx -theory or *behavioural theory* of C . In particular $\text{Th}_=(C)$ denotes the *standard theory* of C . \diamond

Lemma 5.2 For any class $C \subseteq \text{Alg}(\Sigma)$ the following holds:

- (1) $\text{Th}_=(C) = \text{Th}_=(C/\approx)$,
- (2) $\text{Th}_=(\text{Abs}_\equiv(C)) = \text{Th}_=(C)$,
- (3) $\text{Th}_=(\text{FA}_=(C)) = \text{Th}_=(\text{FA}_=(C))$.

Proof: (1) follows from Proposition 3.10.1, (2) follows from Proposition 4.1.1 and (3) is a consequence of Lemma 4.4.1. \diamond

The next proposition shows that for classes of algebras which are constructed by the behaviour operator Beh_\approx or by the abstractor operator Abs_\equiv , \approx -theories can be reduced to standard theories.

Proposition 5.3 For any class $C \subseteq \text{Alg}(\Sigma)$ the following holds:

- (1) $\text{Th}_=(\text{Beh}_\approx(C)) = \text{Th}_=(\text{FA}_\approx(C))$,
- (2) $\text{Th}_=(\text{Abs}_\equiv(C)) = \text{Th}_=(C/\approx)$.

Proof: (1): $\text{Th}_=(\text{Beh}_\approx(C)) = (\text{by definition of } \text{Beh}_\approx) \text{ Th}_=(\text{Abs}_\equiv(\text{FA}_\approx(C))) = (\text{by Lemma 5.2.2}) \text{ Th}_=(\text{FA}_\approx(C)) = (\text{by Lemma 5.2.3}) \text{ Th}_=(\text{FA}_\approx(C))$.

(2): $\text{Th}_=(\text{Abs}_\equiv(C)) = (\text{by Lemma 5.2.2}) \text{ Th}_=(C) = (\text{by Lemma 5.2.1}) \text{ Th}_=(C/\approx)$ \diamond

Proposition 5.3 leads immediately to the following theorem which shows that the \approx -theories of behavioural and abstractor specifications can be characterized by standard

theories. In particular, the first part of Theorem 5.4 shows that the theory of a behavioural specification which is built on top of a standard specification SP is the same as the standard theory of the class of the fully abstract models of SP. Hence we can apply standard proof calculi for proving \approx -theorems over a behavioural specification as soon as we have a (standard) finite axiomatization of the class of the fully abstract models of SP. How such finite axiomatizations can be derived in the case of observable behaviour specifications is studied in [Bidoit, Hennicker 94].

Theorem 5.4 Let $SP = (\Sigma, \text{Cons}, E)$ be a standard specification. Then for (\approx, \equiv) the following holds:

- (1) $\text{Th}_{\approx}(\text{Mod}(\text{behaviour } SP \text{ wrt } \approx)) = \text{Th}_{\approx}(FA_{\approx}(\text{Mod}(SP)))$,
- (2) $\text{Th}_{\approx}(\text{Mod}(\text{abstract } SP \text{ wrt } \equiv)) = \text{Th}_{\approx}(\text{Mod}(SP)/\approx)$.

Proof: Follows from Theorem 4.8 and Proposition 5.3. ♦

Example 5.5 Let **behaviour** SP wrt \approx_{Obs} be an observational behaviour specification. Then $\text{Th}_{\approx}(\text{Mod}(\text{behaviour } SP \text{ wrt } \approx_{\text{Obs}}))$ is called *observational theory of SP* because it consists of all formulas which are observationally satisfied by the observational models of the specification. Since in this case the fully abstract models satisfy an equation $t = r$ if and only if they satisfy all equations $c[t] = c[r]$ for all observable contexts c (cf. Example 3.7.1), one can prove observational theorems by using the standard theory of SP together with the context induction proof technique (cf. [Hennicker 91]). In [Bidoit, Hennicker 94] it is shown how an explicit use of context induction can be avoided.

As a concrete example consider the last two axioms $\text{add}(x, \text{add}(y, s)) \approx \text{add}(y, \text{add}(x, s))$ and $\text{add}(x, \text{add}(x, s)) \approx \text{add}(x, s)$ of the SET specification. Even if these equations were omitted from the specification they would still be observational theorems w.r.t. the observable sorts "bool" and "elem" because for all observable contexts c the equations $c[\text{add}(x, \text{add}(y, s))] = c[\text{add}(y, \text{add}(x, s))]$ and $c[\text{add}(x, \text{add}(x, s))] = c[\text{add}(x, s)]$ can be derived already from the remaining SET axioms. ♦

6 Conclusion

We have presented a framework which clarifies the relationships between the two main approaches to observational semantics. In order to be applicable not only to the observational case but also to other specification formalisms we have introduced a general notion of behavioural specification and abstractor specification and we have seen that there exists a duality between both concepts which allows to characterize behavioural semantics in terms of an abstractor construction and vice versa provided that the underlying equivalence on algebras is factorizable. As an example of a factorizable equivalence we have considered the observational equivalence of algebras w.r.t. a fixed set of observable sorts where arbitrary inputs are allowed for the observable experiments. If we want to deal with a larger class of equivalences including the one of [Nivela, Orejas 88] where only observable inputs are considered all results of this paper can be generalized if we use instead of a family of congruences a family of partial congruences. Thereby a partial congruence over an algebra $A \in \text{Alg}(\Sigma)$ is given by a pair (A_0, \approx_{A_0}) consisting of a subalgebra A_0 of A and a congruence \approx_{A_0} on A_0 . The generalized satisfaction relation is then defined w.r.t. valuations that map variables to elements of A_0 and an algebra A is said to be fully abstract if $A_0 = A$ and $\approx_{A_0} = \approx_A$.

Our semantical characterization theorems lead to proof-theoretic considerations which show that behavioural theories of specifications can be reduced to standard theories of some classes of algebras. In particular, the behavioural theory of a behavioural specification is the same as the standard theory of the class of the fully abstract (standard) models of a specification. Hence we can prove behavioural properties of specifications using standard proof calculi if a finite axiomatization of the class of the fully abstract (standard) models of a specification is provided (see [Bidoit, Hennicker 94]).

Acknowledgement We would like to thank the referees of this paper for valuable remarks. This work is partially sponsored by the French-German cooperation programme PROCOPE, by the ESPRIT Working Group COMPASS and by the German DFG project SPECTRUM.

References

- [Astesiano, Wirsing 89] E. Astesiano, M. Wirsing: Bisimulation in algebraic specifications. In: H. Ait-Kaci, M. Nivat (eds.): Resolution of Equations in Algebraic Structures, Vol. 1, Algebraic Techniques, London, Academic Press, 1-32, 1989.
- [Bernot, Bidoit 91] G. Bernot, M. Bidoit: Proving the correctness of algebraically specified software: modularity and observability issues. Proc. AMAST '91, Techn. Report of the University of Iowa, 139-161, 1991.
- [Bidoit, Hennicker 94] M. Bidoit, R. Hennicker: Proving behavioural theorems with standard first-order logic. Submitted for publication, 1994.
- [Ehrig, Mahr 85] H. Ehrig, B. Mahr: Fundamentals of algebraic specification 1, EATCS Monographs on Theoretical Computer Science 6, Springer, Berlin, 1985.
- [Hennicker 91] R. Hennicker: Context induction: a proof principle for behavioural abstractions and algebraic implementations. Formal Aspects of Computing 4 (3), 326-345, 1991.
- [Knapik 91] T. Knapik: Specifications with observable formulae and observational satisfaction relation. In: M. Bidoit, C. Choppy (eds.): Recent Trends in Data Type Specification, Springer LNCS 655, 271-291, 1991.
- [Milner 77] R. Milner: Fully abstract models of typed λ -calculi. Theoretical Computer Science 4, 1-22, 1977.
- [Nivela, Orejas 88] P. Nivela, F. Orejas: Initial behaviour semantics for algebraic specifications. In: D. T. Sannella, A. Tarlecki (eds.): Proc. 5th Workshop on Algebraic Specifications of Abstract Data Types, Springer LNCS 332, 184-207, 1988.
- [Orejas et al. 91] F. Orejas, M. Navarro, A. Sanchez: Implementation and behavioural equivalence: a survey. In: M. Bidoit, C. Choppy (eds.): Recent Trends in Data Type Specification, Springer LNCS 655, 93-125, 1991.
- [Reichel 81] H. Reichel: Behavioural equivalence -- a unifying concept for initial and final specification methods. In: M. Arato, L. Varga (eds.): Math. Models in Comp. Systems, Proc. 3rd Hungarian Computer Science Conference, Budapest, 27-39, 1981.
- [Reichel 85] H. Reichel: Initial restrictions of behaviour. IFIP Working Conference, The Role of Abstract Models in Information Processing, 1985.
- [Sannella, Tarlecki 85] D. T. Sannella, A. Tarlecki: On observational equivalence and algebraic specification. Proc. TAPSOFT '85, Springer LNCS 185, 308-322, 1985.
- [Sannella, Tarlecki 88] D. T. Sannella, A. Tarlecki: Toward formal development of programs from algebraic specifications: implementation revisited. Acta Informatica 25, 233-281, 1988.
- [Schoett 87] O. Schoett: Data abstraction and correctness of modular programming. Ph. D. thesis, CST-42-87, University of Edinburgh, 1987.
- [Wirsing 86] M. Wirsing: Structured algebraic specifications: a kernel language. Theoretical Computer Science 42, 123-249, 1986.

Extending Pruning Techniques to Polymorphic Second Order λ -Calculus

Luca Boerio

Dipartimento di Informatica, Universita' di Torino
Corso Svizzera 185, 10149 Torino, Italy
lucab@di.unito.it

Abstract. Type theory and constructive logics allow us, from a proof of a formula, to extract a program that satisfies the specification expressed by this formula. Normally, programs extracted in this way are inefficient. Optimization algorithms for these programs have been developed. In this paper we show an algorithm to optimize programs represented by second order typed λ -terms. We prove also that the simplified programs are observational equivalent to the original ones.

1 Introduction

Constructive logics can be used to write the specifications of programs as logic formulas to be proved. Writing programs as constructive proofs of such formulas, is a good attempt to automate programming and program verification. We can extract executable code from constructive proofs, using the Curry-Howard isomorphism of formula-as-types ([8]) or the notion of realizability ([11] and [1]). Following these ideas, some tools have been introduced to help programmers in developing proofs or for the automatic extraction of programs, for example COQ, LEGO, NUPRL ([5] and [6]).

Automatic program extraction has a great problem: often, the extracted code is not efficient. Many attempts have been done to develop methods for the automatic erasing of redundant parts from these programs.

For example, in intuitionistic logic, the formula $\exists x.A(x)$ is interpreted as a pair, whose first element is a term t and whose second element is a proof of $A(t)$. Generally, only the value t of x is needed as program extracted. The proof of $A(t)$, from a computational point of view, is meaningless. Such a proof is not useful for the computation of result. It is only useful to prove that the program meets the specification but this is of use only once and not at every run of the program.

Several algorithms have been defined for erasing redundant code (e.g. Beeson [1] and Mohring [9] that use manual techniques to label and remove redundant code, Takayama [10] which designed an automatic technique that motivated our work). A variant of them are pruning techniques of S. Berardi ([2], [3] and [4]).

Berardi's idea is that, inside the tree representation of an expression, we can find subtrees that are useless for computing the final result. Such parts can be removed in the same way as dead branches in a tree are pruned. Of course the problem is how to find useless subtrees.

In this paper, we show an algorithm to find useless subterms in computations. These subterms can safely be replaced by dummy constants, leading to equivalent terms. The main result of this paper, in particular, is that for each type A and term t of type A , there exists a unique term t' of type A , of minimum length, such that t' is

observational equivalent to t . We'll describe the algorithm that receiving t , gives back this simplest term.

We'll follow the technique used in [4], by defining two order relations ' \leq ', one for types and one for terms. Such relations are an attempt to formalize the previous ideas. $A \leq B$ means that A is simpler than B . In a similar way, if t and u are terms, $t \leq u$ means that t is a simplified version of u . This simpler term has the same input/output behaviour of the original one if the type doesn't change.

In section 2 we introduce some bare notions about the system in which we write our programs. In section 3 we formally define the pruning relation \leq . In section 4 we see that simplifying a term doesn't alter its operational meaning. In section 5 we see that, given a term t , there exists a minimum term t' equivalent to t . In section 6 we show an algorithm to find minimum terms. In section 7 we have conclusions and possible developments.

2 The System F_2

We begin this section with the definition of our system. It is essentially Girard's system F (see [7]), extended with constants for the monomorphic natural numbers and special constants unit and Unit.

The intended purpose of Unit is to denote the type with only one element, denoted by unit. Every expression that we consider useless for the computation of the final result will be substituted by these constants.

Now the formal definitions.

Definition 1.

(i) The language of types of F_2 is inductively defined by

$$T ::= \text{Unit} \mid \text{Nat} \mid X \mid T \rightarrow T \mid \forall X.T$$

where X denotes a type variable, and Nat the constant for natural number type.

(ii) The language of pseudoterms is inductively defined

$$t ::= \text{unit} \mid 0 \mid S \mid \text{rec} \mid x \mid \lambda x : T. t \mid (t t) \mid \Lambda X. t \mid t[T]$$

where 0 and S are the constants for constructing the primitive natural number while rec is the primitive polymorphic constant for the primitive recursion over Nat; x denotes a term variable and $\Lambda X. t$ and $t[T]$ denote type abstraction and type application.

(iii) A context is any finite set of ordered pairs, whose first components are term variables, having pairwise distinct names, and whose second components are types.

(iv) If Γ is a context of the form $\{<x_1 : T_1>, \dots, <x_n : T_n>\}$ $\text{dom}(\Gamma)$ is $\{x_1, \dots, x_n\}$.

As usual, for type system "a la Church", we have rules for well-formedness of terms. A judgement of the form $\Gamma \vdash t : T$, means that the pseudoterm t is a well formed term of type T in the context Γ .

Definition 2.

Let $\Sigma = \{<0:\text{Nat}>, <S:\text{Nat} \rightarrow \text{Nat}>, <\text{rec}:\forall X. \text{Nat} \rightarrow X \rightarrow (\text{Nat} \rightarrow X \rightarrow X) \rightarrow X>, t, t_1 \text{ and } t_2 \text{ be pseudoterms, } x \text{ a term variable, } X \text{ a type variable, } T, A \text{ and } B \text{ types and } \Gamma \text{ a context. The term formation rules are:}$

$$(\text{Unit}) \quad \Gamma \vdash \text{unit} : \text{Unit}$$

$$(\text{Const}) \quad \Gamma \vdash c : A \quad (\text{with } <c:A> \in \Sigma)$$

$$(\text{Var}) \quad \Gamma \vdash x : T \quad (\text{if } <x:T> \in \Gamma)$$

$$\begin{array}{c}
 (\rightarrow I) \quad \frac{\Gamma \cup \{<x:A>\} \vdash t : B}{\Gamma \vdash \lambda x:A.t : A \rightarrow B} \quad (\rightarrow E) \quad \frac{\Gamma \vdash t_1 : A \rightarrow B \quad \Gamma \vdash t_2 : A}{\Gamma \vdash t_1 t_2 : B} \\
 (\forall I) \quad \frac{\Gamma \vdash t : T}{\Gamma \vdash \Lambda X.t : \forall X.T} \text{ (X not free in } \Gamma) \quad (\forall E) \quad \frac{\Gamma \vdash t : \forall X.T}{\Gamma \vdash t[A] : T[X:=A]}
 \end{array}$$

We assume the standard convention for the associativity of operators and for the priorities. We call *atom* any atomic type, constant or variable, and *symbol* any term constant and term variable. We use $\alpha, \beta, \gamma, \dots$ to denote atoms and s, s', s'', \dots to denote symbols. Generally, we use upper case for types and lower case for terms. The symbol ' \equiv ' expresses syntactical equality between expressions.

With $FV(e)$ we indicate the free (type and term) variables of e , while with $FV_1(e)$ and $FV_2(e)$ we respectively denote the free type variables and free term variables of e . As usual, an expression is called *closed* if it hasn't free (type and term) variables. A *substitution* σ is a function from variables to expressions of the language such that, if x is a term variable of type A , $\sigma(x)$ is a term of type A and, if X is a type variable, $\sigma(X)$ is a type. We denote substitutions with σ, τ, \dots . The effect of a substitution over an expression is the replacement of all occurrences of free variables with the corresponding expressions as indicated in the substitution itself. We suppose implicit renaming of bound variables to avoid capture of free variables. A *closed substitution* is a substitution σ such that $\sigma(x)$ is closed for any variable (type and term) $x \in \text{dom}(\sigma)$. If Γ is a context, a substitution σ is said to be a Γ -substitution, if $\text{dom}(\Gamma) \subseteq \text{dom}(\sigma)$. Let e be any expression. We denote $T_2 = \{T \mid T \text{ is a type of } F_2\}$, $T_2^0 = \{T \mid T \text{ is a closed type of } F_2\}$, $\Lambda_2 = \{t \mid t \text{ is a terms of } F_2\}$, $\Lambda_2^0 = \{t \mid t \text{ is a closed terms of } F_2\}$, $E_2 = T_2 \cup \Lambda_2$ and $E_2^0 = T_2^0 \cup \Lambda_2^0$.

The system has β and η rules for terms and types application. We suppose α rules implicit.

Definition 3.

(i) If t, u and f are terms and A a type the elementary reduction rules for F_2 are:

$$\begin{array}{lll}
 (\beta) & (\lambda x:A.t)(u) & \rightarrow_{\beta} t[x:=u] \\
 (\eta) & \lambda x:A.(f x) & \rightarrow_{\eta} f \text{ (with } x \text{ not free in } f) \\
 (B) & (\Lambda X.t)[A] & \rightarrow_B t[X:=A] \\
 (H) & \Lambda X.(f [X]) & \rightarrow_H f \text{ (with } X \text{ not free in } f)
 \end{array}$$

Together with these standard rules there are other ones for the constant rec ; if a is a term of type A and f a term of type $\text{Nat} \rightarrow A \rightarrow A$ then:

$$\begin{array}{lll}
 (\text{rec}0) & \text{rec } [A] \ 0 \ a \ f & \rightarrow_{\text{rec}0} a \\
 (\text{rec}S) & \text{rec } [A] \ (S \ n) \ a \ f & \rightarrow_{\text{rec}S} f \ n \ (\text{rec } [A] \ n \ a \ f)
 \end{array}$$

We write \rightarrow_{rec} for $\rightarrow_{\text{rec}0} \cup \rightarrow_{\text{rec}S}$.

(ii) Let $r \in \{\beta, B, \eta, H, \text{rec}\}$. We use: \rightarrow_r also for the contextual closure of the elementary reduction rule \rightarrow_r ; \rightarrow_1 for the union $\rightarrow_{\beta} \cup \rightarrow_B \cup \rightarrow_{\eta} \cup \rightarrow_H \cup \rightarrow_{\text{rec}}$; \rightarrow_n for exactly n steps of \rightarrow_1 ; $\rightarrow_{\leq n}$ for at most n steps; $\rightarrow_{\geq n}$ for at least n steps. \rightarrow_r^* is used for any finite number of steps of reduction r (included 0). We use $=_r$ for

conversion w.r.t. reduction r . We indicate $=\beta \cup =_B$ with $=\beta_B$ and $\rightarrow\beta \cup \rightarrow_B$ with $\rightarrow\beta_B$, while $=F$ denotes conversion respect all reduction rules together.

We assume the standard definitions of redex and normal form.

An important notion for the analysis of programs is observational equality $=_{obs}$. We can say that two programs are equal from an observational point of view if no observation or test is able to distinguish them. Formally we have:

Definition 4.

Let t, u be two terms such that $\Gamma \vdash t : T$ and $\Gamma \vdash u : T$. We define:

- (i) t and u closed $\Rightarrow \vdash t =_{obs} u$ iff $(\forall \text{closed } f \text{ s.t. } \vdash f : T \rightarrow \text{Nat}) f(t) =_{\beta_B} f(u)$
- (ii) any $t, u \Rightarrow \Gamma \vdash t =_{obs} u$ iff $(\forall \sigma \text{ closed } \Gamma\text{-substitution}) \vdash \sigma(t) =_{obs} \sigma(u)$

We can consider every closed function $f : T \rightarrow \text{Nat}$ as an observation, aiming to discover any difference in the behaviour of t and u . If it is not possible with observations to find a difference between the two terms, we say that they are observationally equivalent.

Definition 5.

Let T be a theory on F_2 . If t and u are terms:

- (i) we write $\Gamma \vdash t =_T u$ if $(t, u) \in T$ and $\text{FV}_2(t) \cup \text{FV}_2(u) \subseteq \text{dom}(\Gamma)$;
- (ii) with $\Gamma \not\vdash t =_T u$ we mean that $\Gamma \vdash t =_T u$ doesn't hold;
- (iii) a theory T is consistent if there exist two terms t, u of same type and context, such that $\Gamma \not\vdash t =_T u$;
- (iv) a theory T is a maximum theory if it is consistent and for each consistent theory T' and for each pair of term t and u , we have $\Gamma \vdash t =_{T'} u \Rightarrow \Gamma \vdash t =_T u$.

Lemma 1.

Observational equivalence is a maximum equational theory for F_2 .

The proof of lemma 1 was developed by Statman and written in an unpublished manuscript of 1986.

Now some properties of F_2 .

Proposition 1.

System F_2 is Church-Rosser and strongly normalizing.

For the proof see e.g. [7].

A term is said *algebraic* if it consists only of: constants of kind $f:\beta$ or $f:\alpha_1 \rightarrow \dots \rightarrow \alpha_n \rightarrow \beta$, where $\beta, \alpha_1, \dots, \alpha_n$ are atomic types, term variables of the form $x:\alpha$, where α is an atom, and applications.

Proposition 2.

Closed algebraic terms of type Nat , in normal form, are only of this shape: $S^k(0)$, for some integer k (if $k = 0 S^k(0)$ is 0).

Proof

Easy by induction.

3 Pruning in F_2

In analogy with [4], we define three relations of \leq , two on *well formed expressions* of the system and one over context. The first relation is over well formed types. We write $A \leq_T B$, if A and B are types. The second relation is defined over contexts. We write $\Gamma \leq_{\text{ctx}} \Gamma'$ if Γ and Γ' are contexts. The third one is defined on well formed terms. We write $t \leq_{\Gamma; A \leq \Delta; B} u$, if t and u are terms s.t. $\Gamma |- t : A$ and $\Delta |- u : B$.

Definition 6.

Let t, t_1, t_2, u_1, u_2 be any terms, A, B, C, D, T any types, c any constant, x, y, X, Y any term and type variables, Γ and Γ' any contexts. The pruning relations " \leq " for types and terms are inductively defined by two deduction systems:

(i) The system \leq_T

$$\begin{array}{lll} (\text{Unit}) \quad \text{Unit} \leq_T T & (\text{Nat}) \quad \text{Nat} \leq_T \text{Nat} & (\text{VAR}) \quad X \leq_T X \\ (\rightarrow) \frac{A_1 \leq_T A_2 \quad B_1 \leq_T B_2}{A_1 \rightarrow B_1 \leq_T A_2 \rightarrow B_2} & (\forall) \frac{A[X:=Z] \leq_T B[Y:=Z]}{\forall X. A \leq_T \forall Y. B} & (*) \end{array}$$

(ii) We say that $\Gamma \leq_{\text{ctx}} \Gamma'$ iff $(\forall x. A) ((x:A) \in \Gamma) \Rightarrow (\exists B) ((x:B) \in \Gamma' \text{ and } A \leq_T B)$.

(iii) The system $\leq_{\Gamma; A \leq \Delta; B}$

$$\begin{array}{ll} (\text{unit}) \quad \frac{\Gamma \leq_{\text{ctx}} \Delta}{\text{unit}_{\Gamma; \text{Unit} \leq \Delta; T} t} & (\text{const}) \quad \frac{\Gamma \leq_{\text{ctx}} \Delta}{c_{\Gamma; A \leq \Delta; A} c} \\ (\text{var}) \quad \frac{\Gamma \leq_{\text{ctx}} \Delta}{x_{\Gamma; A \leq \Delta; B} x} \quad \text{where } (x:A) \in \Gamma \text{ and } (x:B) \in \Delta & \\ (\lambda) \quad \frac{A \leq_T B \quad t_1[x:=z]_{\Gamma; C \leq \Delta; D} \quad t_2[y:=z]}{\lambda x. A. t_1 \Gamma; A \rightarrow C \leq \Delta; B \rightarrow D \lambda y. B. t_2} & (*) \\ & \text{where } \Gamma = \Gamma' \cup \{ (z:A) \} \text{ and } \Delta = \Delta' \cup \{ (z:B) \} \\ (\text{app}) \quad \frac{t_1 \Gamma; A \rightarrow C \leq \Delta; B \rightarrow D \quad t_2 \quad u_1 \Gamma; A \leq \Delta; B \quad u_2}{t_1 u_1 \Gamma; C \leq \Delta; D \quad t_2 u_2} & \\ (\Lambda) \quad \frac{t_1[X:=Z] \Gamma; A \leq \Delta; B \quad t_2[Y:=Z]}{\Lambda X. t_1 \Gamma; \forall X. A \leq \Delta; \forall Y. B \quad \Lambda Y. t_2} & (*) \quad (\text{APP}) \quad \frac{t_1 \Gamma; \forall X. A \leq \Delta; \forall Y. B \quad t_2 \quad C \leq_T D}{t_1[C] \Gamma; A[X:=C] \leq \Delta; B[Y:=D] \quad t_2[D]} \end{array}$$

(*) In expressions involving binders we have to introduce fresh variables z and Z in order to have the relation invariant up to α conversion. Without this caution we would have for example, $\forall X. X \leq_T \forall X. X$, but not $\forall X. X \leq_T \forall Y. Y$.

To simplify the notation, in the rest of the paper we write only \leq , both for \leq_T and for $\leq_{\Gamma; A \leq \Delta; B}$. Since these two relations are defined on different domains, it will be clear from the context of which relation we are talking about.

The relation \leq is a formalization of the idea of simplification of a term. When an expression or a part of an expression is useless at the aim of the computation of the final result, we can replace it with one of the two special constants unit and Unit. So we have a simpler expression.

To better understand how pruning an expression we state:

Proposition 3.

- (i) if A and B are types s.t. $A \leq B$ then
 - either $A \equiv B$
 - or $A \equiv \text{Unit}$
 - or A is obtained from B by replacing some proper subtypes of B with Unit.
- (ii) if t_1 and t_2 are terms s.t. $t_1 \leq t_2$ then
 - either $t_1 \equiv t_2$
 - or $t_1 \equiv \text{unit}$
 - or t_1 is obtained from t_2 by replacing some proper subterms of t_2 with unit
 - or t_1 is obtained from t_2 by replacing some types B, in type applications, with some types A s.t. $A \leq B$
 - or t_1 is obtained from t_2 by replacing the type B of some of its term variables with a type A s.t. $A \leq B$.

All these assertions are up to α conversion.

Remember always that the relation \leq is defined on well formed expression. So if we write $e_1 \leq e_2$ we implicitly assume that e_1 and e_2 are well formed.

Proposition 4.

The pruning relation, \leq , is an order relation.

4 Pruning and Observational Equality

In this section we'll see the main theorem of this paper: pruning is compatible with observation equivalence.

Lemma 2.

If t and u are closed terms in normal form of type Nat, such that $t \leq u$, then $t \equiv u$.

Proof

As in [4].

Lemma 3.

Let x a term variable. Let A and A' types s.t. $A \leq A'$. Let t, t', u and u' terms such that $\Gamma \cup \{\langle x:A \rangle\} \vdash t : B$, $\Gamma' \cup \{\langle x:A' \rangle\} \vdash t' : B'$, $\Gamma \vdash u : A$, $\Gamma' \vdash u' : A'$, for some Γ, Γ', B, B' , and $t \leq t', u \leq u'$. Then $t[x:=u] \leq t'[x:=u']$

Proof

By induction on t .

Lemma 4.

Let t, t', u' be terms. If $t \leq t'$ and $t' \rightarrow_n u'$ with only β reduction steps, then $t \rightarrow_{\leq n} u'$ for some term $u \leq u'$.

Proof

By induction on n , using the previous lemma and the definition of \leq .

Similar properties hold for type substitutions and B reductions.

Lemma 5.

- (i) If A, A', T and T' are types and X is a type variable, such that $A \leq A'$ and $T \leq T'$ then $A[X:=T] \leq A'[X:=T']$.
- (ii) If t and t' are terms, T and T' are types and X is a type variable, such that $t \leq t'$ and $T \leq T'$ then $t[X:=T] \leq t'[X:=T']$.
- (iii) Let t, t' and u' be any terms. If $t \leq t'$ and $t' \rightarrow_n u'$ with only B reduction steps, then $t \rightarrow_{\leq n} u$ for some term $u \leq u'$.

From the previous lemmas the following result follows.

Lemma 6.

Let t, t' and u' be any terms. If $t \leq t'$ and $t' \rightarrow_n u'$, with only β or B reduction steps, then $t \rightarrow_{\leq n} u$ for some term $u \leq u'$.

Now we are ready for the main theorem about the pruning and the observationality.

Theorem 1.

Let t and u terms. If $\Gamma \vdash t : A, \Gamma \vdash u : A$ and $t \leq u \Rightarrow \Gamma \vdash t =_{\text{obs}} u$.

Proof

Using the previous lemmas, the proof is a case analysis of possible contexts and types for t and u .

5 The Minimum Pruning of a Term

We have seen that the pruning relation is an order relation. We can show that for each set of expressions of F_2 there exists the greatest lower bound w.r.t. \leq .

Definition 7.

We inductively define the function $\text{inf} : E_2 \times E_2 \rightarrow E_2$

- (i) for each pair of types,
- (ii) for each pair of terms

(i) Let $T_1, T_2, A, A_1, A_2, B, B_1$ and B_2 be any types, α any atom, X, Y any type variables. We have:

$$\begin{aligned} \text{-inf}(\alpha, \alpha) &= \alpha \\ \text{-inf}(A_1 \rightarrow B_1, A_2 \rightarrow B_2) &= \text{inf}(A_1, A_2) \rightarrow \text{inf}(B_1, B_2) \\ \text{-inf}(\forall X. T_1, \forall Y. T_2) &= \forall Z. \text{inf}(T_1[X:=Z], T_2[Y:=Z]) \quad (Z \text{ fresh type variable}) \\ \text{else} \\ \text{-inf}(A, B) &= \text{Unit} \end{aligned}$$

(ii) Let $t, t', t_1, t_2, u, u', u_1$ and u_2 be any terms, c any constant, X, Y any type variables, A, B any types, x, y any term variables then:

$$\begin{aligned} \text{-inf}(c, c) &= c \\ \text{-inf}(x, x) &= x \\ \text{-inf}(\lambda x: A. t, \lambda y: B. u') &= \lambda z. \text{inf}(A, B). \text{inf}(t[x:=z], u'[y:=z]) \quad (\text{where } z \text{ is a fresh term variable}) \\ \text{-inf}(t_1 t_2, u_1 u_2) &= \text{inf}(t_1, u_1) \text{inf}(t_2, u_2) \\ \text{-inf}(\Lambda X. t, \Lambda Y. u') &= \Lambda Z. \text{inf}(t[X:=Z], u'[Y:=Z]) \quad (\text{where } Z \text{ is a fresh type variable}) \\ \text{-inf}(t'[A], u'[B]) &= \text{inf}(t', u')[\text{inf}(A, B)] \end{aligned}$$

```

else
-inf(t,u) = unit

```

All these assertions are up to α renaming.

Extending these definitions to contexts we have:

Definition 8.

For each pair of contexts exists the \inf_{ctx} and, if Γ and Δ are contexts, we define $\inf_{\text{ctx}}(\Gamma, \Delta) = \{\langle x:C \rangle \mid \langle x:A \rangle \in \Gamma, \langle x:B \rangle \in \Delta \text{ and } C = \inf(A, B)\}$

Now we can state, without proof

Lemma 7.

(i) If e_1 and e_2 are both types or both terms and Γ_1 and Γ_2 are contexts then $\inf(e_1, e_2)$ is the g.l.b. of e_1 and e_2 w.r.t. \leq while $\inf_{\text{ctx}}(\Gamma_1, \Gamma_2)$ is the g.l.b. of Γ_1 and Γ_2 w.r.t. \leq_{ctx} .

(ii) The sets T_2 and Λ_2 are lower semilattices w.r.t. \leq .

As consequence, we have the theorem which puts in relation deduction with pruning.

Theorem 2.

If t and u are terms s.t. $\Gamma \vdash t : A$ and $\Delta \vdash u : B$ and there exists a term z s.t. $t \leq z$ and $u \leq z$, then

$$\inf_{\text{ctx}}(\Gamma, \Delta) \vdash \inf(t, u) : \inf(A, B)$$

Now we introduce two structures useful for the optimization algorithm that we'll define in the next section.

Definition 9.

For each term t , s.t. $\Gamma \vdash t : T$ we define:

- (i) $LE(t) = \{t' \mid t \leq t'\}$
- (ii) $CLE(t) = \{t' \mid t \leq t' \text{ and } \Gamma \vdash t' : T \text{ and } \Gamma' \subseteq \Gamma\}$

If t is a term, the set $LE(t)$ (*less equal t*) is the set of all the terms that we can obtain from t replacing some parts by the special constants, while the set $CLE(t)$ (*contextual less equal*) is the set of simplified version of t with same type and context and so equivalent to t itself.

Proposition 5.

Given any term t we have:

- (i) $LE(t)$ is a finite complete lower semilattice w.r.t. \leq
- (ii) $CLE(t)$ is a sub semilattice of $LE(t)$

Proof

- (i) A simple consequence of lemma 7.
- (ii) Easy, generalizing theorem 2 and remembering that every term $t' \in CLE(t)$ has the same type of t and has a context included in Γ .

Definition 10.

Let t any term. We denote the minimum element of $CLE(t)$ as $FL_2(t)$.

Proposition 6.

$\text{Fl}_2(t) =_{\text{obs}} t$.

Proof

From theorem 2, since $\text{Fl}_2(t) \leq t$.

6 An Algorithm for Finding Minimum Prunings

In this section we show an algorithm to find the term $\text{FL}_2(t)$, defined in last section. In order to compute FL_2 we use the technique of C. Mohring based on marking of types (see [9]). A mark is a label that we put over an atomic type, constant or variable. Marks are of two kind: 'r' and 'c'. The former means that a term or subterm whose type has such mark is redundant, i.e. useless for the computation of final result. Instead the latter mark means that the expression *may be* useful for the computation of the result. When an expression is entirely marked we can remove the parts marked 'r'. We let parts marked 'c' while we replace redundant types by Unit and the corresponding (redundant) terms by unit. So there is a strong correspondence between Mohring's manual technique of type labelling and our extended syntax with constants denoting useless expression. The marking technique will be used to compute a 'minimum' marking that corresponds to a 'minimum' term (with respect to the pruning relation). The underlying computational structure is the abstract syntax tree of a term. We use the *fully decorated tree*, in which associated with every node there is the type of the subterm individuated by this node. Now, we introduce some definitions and terminology about trees and markings.

Definition 11.

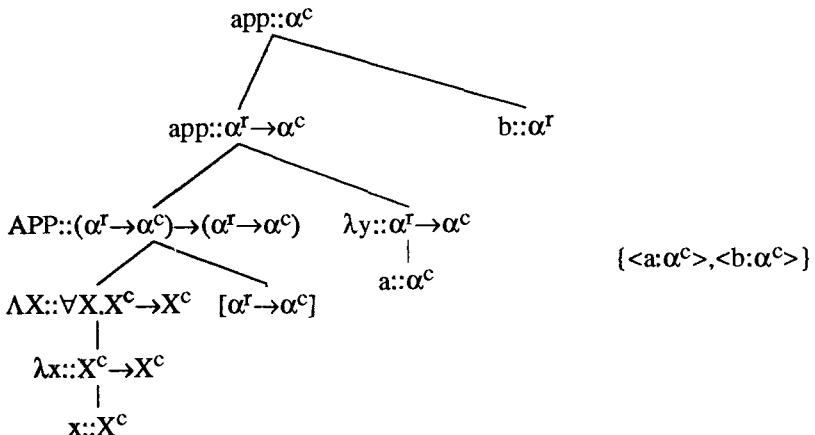
Let t be a term such that $\Gamma \vdash t : T$

- (i) The fully decorated tree of t , $\text{FDT}(t)$, is the syntax tree of t in which each node that identifies a subterm is decorated by the type of the subterm itself.
- (ii) The fully decorated version of $\Gamma \vdash t : T$, $\text{FDV}(\Gamma \vdash t : T)$, is defined as the ordered pair $\langle \Gamma, \text{FDT}(t) \rangle$, formed by the context Γ and the fully decorated tree of t .
- (iii) $\text{Atoms}(\Gamma, t)$, the set of occurrences of atoms of $\text{FDV}(\Gamma \vdash t : T)$, is formed by every occurrence of atom in the types of the variables in the context Γ with every occurrence of atom in the types that decorate the nodes of $\text{FDT}(t)$ and the occurrences of atoms of applied types in type applications of t .
- (iv) Let Lab be the set formed by the two labels 'r' and 'c', namely $\text{Lab} = \{\text{'r'}, \text{'c'}\}$. A marking M of $\text{FDV}(\Gamma \vdash t : T)$ is a map from $\text{Atoms}(\Gamma, t)$ to Lab . When we say a marking M of t , we mean the restriction of the map M to $\text{FDT}(t)$. Note that we can identify a marking of a FDV with the set $C \subseteq \text{Atoms}(\Gamma, t)$ of occurrences of atoms that are marked by 'c'.

We show these concepts with an example

Example.

Let $\text{term}_1 = ((\lambda X. \lambda x : X. x)[\alpha \rightarrow \alpha](\lambda y : \alpha. a))b$ with α an atom, a and b free term variables of type α . We can deduce $\{<a : \alpha>, <b : \alpha>\} \vdash \text{term}_1 : \alpha$. The FDV for this judgement, with an example of marking for it, is:



We call this marking M_1 .

Not all the markings are useful or meaningful for the pruning. We single out some kinds of markings for their use in optimization.

Definition 12.

Let t be a term. A marking M of t is *canonical* if for each node V of $FDT(t)$ no atom in the types associated to descendent nodes of V is marked with a 'c', when the type associated to V is completely redundant, i.e. all of its atoms are marked with label 'r'.

Now we define the map that gives the correspondence between marking and pruning.

Definition 13.

Given a fully decorated syntax tree for a term t and a canonical marking M for it, we denote $Simplify(M, t)$ the term obtained replacing every maximum subtype in t , totally marked by 'r', with Unit and every maximum subterm of t , whose type is totally marked by 'r', with unit.

Now, we can define others kinds of markings.

Definition 14.

Let t be a term s.t. $\Gamma \vdash t : T$,

- (i) A marking M on t is *consistent* iff $Simplify(M, t)$ is a well formed term w.r.t. some context $\Gamma \leq_{ctx} \Gamma$.
- (ii) A marking M is *saturated* iff it is consistent and canonical.

Observe that, to have consistency in a marking of t , any two atoms matched during the typechecking of t have to be marked in the same way.

As already said, there is a strong correspondence between marking and pruning. This correspondence is expressed by the map $Simplify$. It is an isomorphism from saturated markings and the expressions of the system in which: 1) there are no non-atomic types whose atoms are just Unit; 2) the only term of type Unit is unit. With this choices, we can prove that if t is a term, for each saturated marking M there exists one and only one pruned term $t' \leq t$ such that $Simplify(M, t) = t'$.

For example the marking M_1 is saturated. Applying Simplify to it, we have $\text{Simplify}(M_1, \text{term}_1) = ((\Lambda X. \lambda x : X. x)[\text{Unit} \rightarrow \alpha](\lambda y : \text{Unit}. a))\text{unit}$ that is also the minimal element of $\text{CLE}(\text{term}_1)$, namely $\text{FL}_2(\text{term}_1)$.

We can define also an order on markings.

Definition 15.

Let t be a term s.t. $\Gamma \vdash t : T$. Let M and M' be markings for the $\text{FDV}(\Gamma \vdash t : T)$. We say:

$M \leq M'$ iff each 'c' assigned by M is also assigned by M' .

We can prove that the two ordering correspond to each other. If t is a term and M, M' markings over t , then $M \leq M'$ iff $\text{Simplify}(M, t) \leq \text{Simplify}(M', t)$.

Obviously, the minimum marking has all 'r'. In this way, the whole term is replaced by unit and we obtain a term which is not equivalent to the original. We look for a term with same type and same context. So we use an *initial marking* M_0 that brought with it these conditions.

Definition 16.

Given a $\text{FDV}(\Gamma \vdash t : T)$, the initial marking M_0 for it, is the map that assigns 'r' to every atom in $\text{Atoms}(\Gamma, t)$, excluding: 1) the atoms in the type associated to the root, namely *the type of the term*; 2) the atoms in the type of variables inside the context.

This initial marking is canonical but inconsistent. The aim of our algorithm is, to find a minimum consistent marking M' that is greater (respect to \leq) than the initial marking M_0 . This operation is called *saturation* of a marking. In this section we call *binder* both $\lambda x : A$ and ΛX in term and pair $\langle x : A \rangle$ in context.

Now, we can write our optimization algorithm.

Definition 17.

Optimization Algorithm

Input: A term t , a context Γ and a type T such that $\Gamma \vdash t : T$.

Output: $\text{FL}_2(t)$.

Perform in sequence the following steps:

- Build the $\text{FDV}(\Gamma \vdash t : T)$.
- Build the initial marking M_0 .
- Saturate M_0 , obtaining M' .
- Apply $\text{Simplify}(M', t)$.

The interesting part of the algorithm is then in the saturation procedure. It consists of two parts. In the first part we build a structure, the adiacence graph that is used to connect the atoms that have to be marked with the same label. So the edges of the graph are a different way to describe consistency condition. If two objects are linked, it means that they have been matched during the type checking procedure. In the second part, the labels 'c', starting from the type T of the term, flow along the paths of the adiacence graph. When this flow is completed the algorithm stops and the resulting marking is consistent.

Definition 18.**Saturation Algorithm**

Input: A FDV($\Gamma \vdash t : T$) and M_0 for it.

Output: The minimum saturated marking M' s.t. $M_0 \leq M'$

- Build the adiacence graph of FDV($\Gamma \vdash t : T$).
- Propagate the labels to achieve consistency.

Now we see the algorithm for the construction of the adiacence graph. In this step, we build the structure used for obtaining the consistency in the marking.

Definition 19.

(i) Given a FDV($\Gamma \vdash t : T$), we call adiacence graph, AG($\Gamma \vdash t : T$), the graph so defined:

- 1) The set Nod of nodes is formed by hte union of:
 - a) Atoms(Γ, t)
 - b) the set of occurrences of term variables of FDT(t)
 - c) the set of binders in the FDT(t) and in the context
 - d) the set of all types of FDT(t)
 - 2) The set Edg of edges is formed by four kinds of edges connecting different kinds of nodes:
 - a) *normal edges* connect pairs of atoms in types
 - b) *bind edges* connect free variables with their binders in the context and bound variables with their binders in term
 - c) *broadcast edges* link type variables with types
 - d) *bind broadcast edges* link types with types in square brackets.
 - 3) Every part of the FDV($\Gamma \vdash t : T$), matched during the typechecking, is linked by a related edge.
- (ii) Given a FDV($\Gamma \vdash t : T$), its augmented tree, Tedge($\Gamma \vdash t : T$), is obtained from the FDV adding the AG($\Gamma \vdash t : T$) on it.

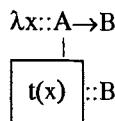
Definition 20.**Adiacence Graph Construction Algorithm**

Input: A FDV($\Gamma \vdash t : T$) and M_0 for it.

Output: The Tedge($\Gamma \vdash t : T$) of FDV.

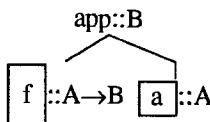
Perform the following steps:

- 1) Link,with a bind edge, every free term variable to its binder in the context Γ
- 2) For each λ -abstraction node



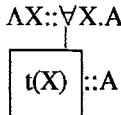
of the tree, link corresponding atoms in the two occurrences of B with normal edges and link every occurrence of the bound variable x in t to its binder λx with a bind edge.

- 3) For each term application node



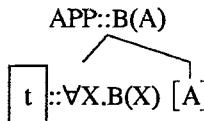
of the tree, link corresponding atoms in the two occurrences of A and in the two occurrences of B with normal edges.

4) For each Λ -abstraction node



of the tree, link corresponding atoms in the two occurrences of A with normal edges.

5) For each type application node



of the tree, link every occurrence of X in $B(X)$ to the corresponding occurrences of A in $B(A)$ with a broadcast edge. Link corresponding atoms in the two occurrences of B, that are not X or atoms of A, with normal edges. Link every occurrence of A in $B(A)$ to the occurrence of A in $[A]$ with a bind broadcast edge.

Now the second step. With this structure, the consistency of a marking is equivalent to the following conditions:

- (1) for each pair of atoms connected with a normal edge they are marked in the same way;
- (2) for each free or bound term variable $x:A$ in t , if A has at least one mark 'c' and $\langle x:A \rangle \in \Gamma$ or $\lambda x:A$ is the binder of x, corresponding atoms in the two occurrences of A are marked in the same way;
- (3) for each type variable connected to a type by a broadcast edge, either the variable and the atoms of the type are marked 'r' or the variable and at least one atom of the type are marked 'c';
- (4) for each occurrence of type A, connected to a type A in square brackets, by a bind broadcast edge, if its marking contains at least one 'c' then corresponding atoms in the two occurrences of A are marked in the same way.

There is still something to say. In our system the type of the constants cannot be simplified. So there is only a term strictly less than a constant, namely unit. This imply that either a constant is removed or is totally used. So we have to add a fifth case to the procedure, for the treatment of constants.

Definition 21.

Marking Propagation Algorithm

Input: The $T_{edge}(\Gamma |- t : T)$ of FDV and the initial marking M_0 .

Output: The minimum saturated marking M' s.t. $M_0 \leq M'$.

Repeat one of the following steps **until** no more step can be executed:

- (1) Take any normal edge such that at least one atom is marked 'c'. Then mark the other atom 'c' and remove the edge;
- (2) or take any bind edge such that the free or bound variable has at least one 'c' in the marking of its type. Let $x:A$ this variable and $\langle x:A \rangle \in \Gamma$ or $\lambda x:A$ its binder. Then remove the bind edge and link with one normal edge each corresponding atom in the two occurrences of A;
- (3) or take a broadcast edge such that either the type variable is marked 'c' or the type has an atom marked 'c'. Then, in the first case, mark last atom of the type

with 'c'. In the second case mark the type variable with 'c'. In both cases remove the edge;

(4) or take a bind broadcast edge such that the type not inside the square brackets has at least one 'c' in its marking. Let A this type. Link each corresponding atom in the two occurrences of A with a normal edge. Remove the bind broadcast edge;

(5) or take a constant $c:A$ such that at least one of its labels is 'c'. Then label with 'c' each atom of A.

This part ends the description of our optimization algorithm. We can prove that saturation algorithm always stops with the minimum consistent marking, namely the one with less 'c' in it, s.t. the type and the context are totally marked with 'c'. Owing to the correspondence between pruning and marking, this leads to the minimal pruning among the ones which respect the type and the context of the original term.

Theorem 3.

Let t be a term s.t. $\Gamma \vdash t : T$. The saturation algorithm computes the saturation of M_0 , i.e. the minimal marking M' s.t. $M_0 \leq M'$ and M' is saturated.

Theorem 4.

Let t be any term. Simplify is an isomorphism between saturated markings of t and the set of terms $t' \leq t$, whose syntax respects the rules: 1) don't exist non-atomic types whose atoms are just Unit; 2) the only term of type Unit is unit.

As consequence of the two previous theorems, we have the correctness of our optimization algorithm.

Theorem 5.

Given a term t such that $\Gamma \vdash t : T$ for some type T and context Γ , the optimization algorithm computes $Fl_2(t)$, i.e. the minimal term $t' \leq t$ whose type is T and whose context is $FV_2(t)$.

7 Conclusions and Future Works

In this paper, we have described a first step toward a simplification of expressions in F_2 . Every time we find a subexpression useless for the final result, we replace it by a unit. New expressions are shorter but may contain a lot of instances of constant unit. We are developing another step, i.e. the elimination of these instances. In this way we save more time and space during the evaluation of the expressions.

In future, we want also introduce the notion of Harrop type in our system. Informally, a type H is Harrop if all terms of such type are equivalent. So, for each Harrop type H , we can define a canonical constant c_H and replace each term t of type H by this constant.

Another idea is to introduce Kinds in our system. Kinds may be useful for pruning type variables. If a λ -binder in a term binds a term variable x of type A never used in the body of the abstraction, we can prune $\lambda x:A$ to $\lambda x:\text{Unit}$. For a Λ -binder that binds a type variable never used, we can do nothing. Introducing a special Kind UNIT, we can mark this useless abstraction, using this new constant, and in future remove such an abstraction.

Finally, we have to remember that, our techniques are useful above all for automatically generated programs. In such programs there may appear large parts of

redundant code. Normally, for code generated by hand, the best optimizer is the programmer's head.

Acknowledgements

I want to thank my supervisor M. Coppo for his help in recent years and for his valuable remarks and suggestions about this paper. Above all I thank S. Berardi since without his foundational work, his encouragement and discussions about pruning this paper would never have been written.

Bibliography

- [1] M. Beeson, *Foundations of Constructive Mathematics*, Berlin, Springer-Verlag, 1985
- [2] S. Berardi, *Extensional Equality for Simply Typed λ -calculi*, Technical Report, Turin University, 1993.
- [3] S. Berardi, *A canonical Projection between Simply Typed λ -calculi*, Technical Report, Turin University, 1993.
- [4] S. Berardi, *Pruning Simply Typed λ -terms*, Technical Report, Turin University, 1993.
- [5] R. L. Constable et al., *Implementing Mathematics with the Nuprl Proof Development System*, Prentice-Hall, 1986.
- [6] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, B. Werner, *The Coq Proof Assistant- User's Guide*, INRIA - Rocquencourt, 1983.
- [7] J.-Y. Girard, *Interpretation Fonctionnelle et Elimination des Coupures de l'Arithmetique d'Ordre Superieur*, These de Doctorat d'Etat, Soutenue le 26 Juin 1972.
- [8] W.A. Howard, *The Formulae-as-Types notion of Construction*, in 'Essays on Combinatory Logic, Lambda Calculus and Formlism', Eds J. P. Seldin and j. R. Hindley, Academic Press, 1980.
- [9] C. Paulin-Mohring, *Extracting F_ω 's Programs from Proofs in the Calculus of Constructions*, In: Association for Computing Machinery, editor, Sixteenth Annual ACM Symposium on Princiles of Programming Languages, 1989.
- [10] Y. Takayama, *Extraction of Redundancy-free Programs from Constructive Natural Deduction Proofs*, Journal of Symbolic Computation, 1991, 12, 29-69
- [11] A. S. Troelstra, *Mathematical Investigation of Intuitionistic Arithmetic and Analysis*, Lecture Notes in Mathematics, 344, Springer-Verlag, 1973

λ -definition of Function(al)s by Normal Forms*

Corrado Böhm¹, Adolfo Piperno¹, Stefano Guerrini²

¹ Dipartimento di Scienze dell'Informazione, Università di Roma "La Sapienza",
Via Salaria 113, 00198 Roma, Italy,
e-mail: {boehm,piperno}@dsi-next1.ing.uniroma1.it

² Dipartimento di Informatica, Università di Pisa,
Corso Italia 40, I-56100 Pisa, Italy,
e-mail: guerrini@di.unipi.it

Abstract. Lambda-calculus is extended in order to represent a rather large class of recursive equation systems, implicitly characterizing function(al)s or mappings of some algebraic domain into arbitrary sets. Algebraic equality will then be represented by $\lambda\beta\delta$ -convertibility (or even reducibility). It is then proved, under very weak assumptions on the structure of the equations, that there always exist solutions in normal form (Interpretation theorem). Some features of the solutions, like the use of parametric representations of the algebraic constructors, higher-order solutions by currying, definability of functions on unions of algebras, etc., have been easily checked by a first implementation of the mentioned theorem, the CuCh machine.

1 Introduction

Combinatory logic [17] and λ -calculus [16] are different logic theories. Since there is still a one to one correspondence between a combinator and a closed λ -term, for the sake of simplicity we will refer to λ -terms most of the time.

A normal form (nf) is a λ -term irreducible respect to any β (η) reduction rule. Term reduction being the theoretic counterpart of computation, Church and its scholar Kleene proved the equivalence between λ -definability and recursive function theory finding out nf's representing any natural number or any recursive function [22].

On the other side Curry and Turing had a more liberal point of view on computability, in that computation shall not imply termination; e.g., Turing wrote of a machine computing the digits of π [30]. They discovered fixed point combinators \mathbf{Y}_t and \mathbf{Y}_c , both without nf, to define partial functions introduced, e.g., by the μ -operator. The nf's used by Kleene to represent primitive or even general recursive functions were particularly intricate and not perspicuous.

In the mid 60's a small group of people, Wagner, Strong, and others [32, 28] tried to generalize recursive function theory to any type of data by Uniformly Reflexive Structures (URS), based on entities similar to λ -terms.

In the early 70's the treatment of recursive functions by fixed point combinators appeared more fascinating than the other approach, since they represent

* This work has been partially supported by grants from ESPRIT BRA 7232 working group "Gentzen" and from MURST 40% (Italy).

the idea of universal iterator, a finite object that can iterate functions an infinite number of times. Scott, Wadsworth and others were then able to construct the denotational semantics of programming languages based on fixed point theory and λ -calculus.

Simultaneously Wadsworth, Welch and others developed the notions of “head normal form” and “finite development,” namely the $\lambda\Omega$ -calculus. Still in the 70’ the ADJ group and several other researchers developed an algebraic basis to programming, the “algebraic data types”.

Backus [3] proposed to change the programming style a la Von Neumann, using variables and assignment statements, into an applicative style FP or FPP (similar to LISP) that avoided the use of variables.

Böhm [6] proved that FP was embeddable into combinatory logic and then our research group became interested to the embedding of algebraic data types and relative mappings of algebras (into another set) by λ -terms. A method to represent any type of term algebras and functions “iteratively” defined on that algebras, using second order typed nf’s was introduced by [11]. This was a first incomplete but meaningful improvement on [22] and on URS. [7] and [8] extended the class of iterative functions, by means of two different methods, to treat primitive recursive and mutual iterative schemes. A remarkable result, preserving Church numeral system, exhibits a normal form for primitive recursive functionals [26].

However, two questions remained still unanswered: 1) Was the typing really necessary? 2) Could the same results be achieved for general schemes of recursion, defining any partial functions on any data structures? [9] answered positively both questions. The use of Böhm-tree [14, 15, 4] proved that the combinators representing the constructors of any homogeneous terms algebras are also a basis for the full combinatory logic and indeed for nf’s. There is in addition a one to one mapping * transforming the constructors into new constructors, therefore algebras into *-algebras, on which a class of equation schemes defining partial recursive functions admits solutions in nf.

[10] illustrated the interdependence between equation schemes and the choice of λ -terms representing zero and the successor function, to obtain nf for the solution of some schemes (generalizing [24]). The schemes examined were iterative, primitive recursive, general recursive and, for the last one, double recursion has been reduced by currification to the single one. The extension of the method of definition to term algebras of arbitrary data structure remains unanswered, as well as the treatment of mutual recursion schemes.

[5] defines rewriting systems, called “canonical and algebraic”, and describes a Böhm–Piperno technique to obtain a definition in nf of a self-interpreter and of a reducer of a gödelization of the λ -calculus into itself.

The present paper shows how to expand the class of canonical systems so that our treatment is still valid. In addition, we list some attractive features of our nf solutions, otherwise lacking using fixed point combinators, and of interest for people looking for a concrete application of the theorems here presented.

Without embarking into a deep philosophical treatment, we would like to

convince the reader that the ideas behind our success in finding nf representing recursive functions or functionals on algebraic data can be made at least as popular as those of “structured” or “object oriented” programming. This is the aim of the following introduction. Let us begin quoting J.Shoenfield in connection with formal systems (page 2 of [27]):

...if we choose the language for expressing the axioms suitably, then the structure of the sentence will reflect to some extent the meaning of the axiom.

Our language is the λ -calculus. We must model algebraic expressions containing data as well as previously or newly defined functions. Our aim is to eliminate recursion from the definition of some function. The only possible way is to move inductive definitions from mappings to be defined into previously defined functions, i.e., the constructors of the domains of mappings. We may then talk of “data driven programming”, an idea that extrapolates usual concepts from object oriented programming. Applying Shoenfield’s recommendation we may choose for the algebraic language a syntax underlining the mentioned dichotomy between data and functions and simplifying λ -definitions. For the sake of this introduction, constructors will be written in prefix notation, whereas functions needing a recursive definition will be in suffix notation (if unary) and infix notation (if binary or n+2-ary). Let us associate to Booleans (**B**), natural Integers (**N**) and Lists of elements of a set A (L_A), the following features of their constructors: {name : arity, ...}. Then we have:

$$\mathbf{B} : \{\text{True} : 0, \text{False} : 0\}, \mathbf{N} : \{0 : 0, 1+ : 1\}, L_A : \{\text{nil} : 0, \text{cons} : 2\}.$$

We will give three examples of definition of functions: an explicit one for the ternary function if-then-else (ite), an iterative definition for the addition function (+), and a primitive recursive definition for the termial function (?) [23] whose intuitive definition is

$$n ? = 0 + 1 + \dots + (n - 1) + n.$$

Using our syntax we can write

$$\begin{aligned} \text{True ite } x \ y &= x \\ \text{False ite } x \ y &= y \end{aligned}$$

translating into the λ -calculus True and False it appears natural to consider the last two equations as definitions of True and False as combinators (and to treat ite as a variable). This poses the problem of defining a link between a constructor and a function, essentially the need to have a universal λ -definition for a constructor of given arity and simultaneously a systematic way to replace a constructor with a combinator related to the form of the equation. The answer will be found in the paper and, for the moment, we will ignore this problem. Let us write down two properties of the addition:

$$\begin{aligned} 0 + n &= n \\ (1 + m) + n &= 1 + (m + n) \end{aligned}$$

This system of equalities can be easily transformed into a recursive definition of the function $+$ by the equality between the functions $1+$ and $+$:

$$1 + x = 1+ x.$$

By replacement we obtain the recursive definitions

$$\begin{aligned} 0 + n &= n \\ 1+ m + n &= 1+ (m + n) \end{aligned}$$

that become explicit definitions of the combinators 0 and $1+$ considering $+, m, n$ as variables. We must notice: a) that positive integers are constructed as follows:

$$1 = 1+ 0, 2 = 1+ 1, 3 = 1+ 2, \dots$$

b) In the second equation at the rhs $1+ (m+n)$ cannot be reduced since a property of combinator weak reduction is that reduction can take place only if the number of arguments of the combinator is greater or equal to that one appearing in the definition (here 3). Thus, the result of computing $3+n$ is $1+ (1+ (1+ n))$, the result remaining valid if, before or after the computation, we will replace n by any non negative integer.

A tentative primitive recursive definition of $?$ like

$$\begin{aligned} 0 ? &= 0 \\ (1+ n) ? &= (1+ n) + (n ?) \end{aligned}$$

could be translated into an explicit λ -definition only if we possess some delta-rules to define addition or if we can consider $+$ as a predefined combinator. We can obviously form a system of four simultaneous equations and try to solve it, but we would have the same difficulty encountered above. An additional difficulty would arise in mutual simultaneous recursion.

The next section will solve all these difficulties. To spare the efforts of the reader we will return to prefix notation for all kinds of functions. Alert readers will discover factual infix notation hidden during β -reduction of some terms.

2 Recursive Equations and λ -calculus

As usual, we shall consider the set Λ of terms of the λ -calculus to be described by the following BNF, where a and x range over denumerable sets of constants and variables, respectively:

$$L ::= a \mid x \mid (\lambda x. L) \mid (L_1 L_2). \quad (1)$$

Let Σ be a set of function symbols from a given signature. $\Lambda(\Sigma)$ denotes the set of *extended lambda terms* with symbols from the signature Σ . To be precise $\Lambda(\Sigma)$ can be defined by adding the following clause to the clauses (1) for the formation of lambda terms: if $t_1, \dots, t_n \in \Lambda(\Sigma)$ and $f \in \Sigma$ is an n -ary function symbol, then $f(t_1, \dots, t_n) \in \Lambda(\Sigma)$. Note that $\text{Ter}(\Sigma) \subseteq \Lambda(\Sigma)$ where $\text{Ter}(\Sigma)$ is the set of first order terms with signature Σ .

Definition 1. Let \mathcal{E} be a set of equations in the extended λ -calculus $\Lambda(\Sigma)$. We say that \mathcal{E} is *canonical* if the function symbols in Σ can be partitioned in two disjoint subsets $\Sigma = \Sigma_0 \cup \Sigma_1$ so that, letting $\Sigma_0 = \{c_1, \dots, c_r\}$ and $\Sigma_1 = \{f_1, \dots, f_k\}$, each equation $t = t'$ of \mathcal{E} has the form

$$f_i(c_j(x_1, \dots, x_m), y_1, \dots, y_n) = b_{i,j} \quad (2)$$

where $f_i \in \Sigma_1$, $c_j \in \Sigma_0$, $b_{i,j} \in \Lambda(\Sigma)$ is a term depending on i and j , $n, m \geq 0$ and the variables $x_1, \dots, x_m, y_1, \dots, y_n$ are all distinct (left-linear).

We call the elements of Σ_0 *data constructors* and those of Σ_1 *programs*. We say that \mathcal{E} is *complete* if for all $f_i \in \Sigma_1$ and $c_j \in \Sigma_0$, \mathcal{E} contains exactly one equation of the form (2).

Notice that we allow some lambda abstractions and applications to appear on the right-hand-sides of equations of a canonical system but not on the left-hand-sides.

Important examples of data are natural numbers, with constructors *zero* and *succ* and parametric lists with constructors *cons* and *nil*.

In order not to interdict concrete applications of λ -calculus, we assume constants to be integers (the set of integers will be denoted by $\mathbb{N}_\delta = \{0, 1, 2, \dots, 10, 11, \dots\}$) and booleans (notation $\mathbf{B}_\delta = \{\text{True}, \text{False}\}$) together with strict elementary functions on such constants, called δ -operators; the notion of reduction associated to them (δ -reduction) will be intended without any special notation. Prefix applicative notation will be used for δ -operators. As an example, conditional expressions will be defined by means of the δ -operator $\text{ite} : \mathbf{B}_\delta \rightarrow \Lambda$ such that $\text{ite } \text{True} = \mathbf{K} \equiv \lambda xy.x$ and $\text{ite } \text{False} = \mathbf{O} \equiv \lambda xy.y$.

It comes out that we allow an ambiguous representation of natural numbers, e.g. 3 and $\text{succ}(\text{succ}(\text{succ zero}))$, and succ 2, too. The coexistence of such different representations will be clarified in section 3.1.

The following definition imposes some restrictions over right hand sides of canonical systems of equations.

Definition 2. Let \mathcal{E} be a set of equations in the extended λ -calculus $\Lambda(\Sigma)$. We say that \mathcal{E} is *safe* if it is canonical and moreover the following conditions hold for all $t = t' \in \mathcal{E}$:

- (i) t' is a $\beta\delta$ -normal form;
- (ii) $\forall f \in \Sigma_1. \forall T_1, \dots, T_n \in \Lambda(\Sigma). f(T_1, \dots, T_n)$ occurs in $t' \Rightarrow T_1$ has no initial abstractions;
- (iii) $\forall c \in \Sigma_0. \forall T_1, \dots, T_n, T_{n+1} \in \Lambda(\Sigma). c(T_1, \dots, T_n)T_{n+1}$ never occurs in t' ;
- (iv) $\forall f \in \Sigma_1. \forall T_1, \dots, T_n \in \Lambda(\Sigma). f(T_1, \dots, T_n)$ occurs in $t' \Rightarrow T_1 \notin \mathbb{N}_\delta$.

Remark. Some of the constraints introduced over canonical systems to make them safe could also be obtained introducing types over the specification language $\Lambda(\Sigma)$. Indeed (ii) and (iii) in definition 2 are implied, in a typed scenario, by the imposition of a basic type on T_1 and $c(T_1, \dots, T_n)$, respectively. Notice that, since we will interpret $\Lambda(\Sigma)$ in the *pure* λ -calculus (see sect.3), such interpretation will be independent of the choice of a particular type system for the specification language.

As pointed out by one of the referees, the restriction (ii) can be eliminated introducing a new Σ_1 symbol f' , a new Σ_0 symbol c' , an equation

$$f'(c', x_1, \dots, x_n) = f(x_1, \dots, x_n)$$

and replacing all terms of the form $f(T_1, \dots, T_n)$ in right-hand sides in which T_1 has initial abstractions by $f'(c', T_1, \dots, T_n)$. This shows that the mentioned restriction does not cause any loss in the expressive power of the language.

Finally, the restriction (iv) is due to technical reasons, only. Also, it does not cause any loss in the expressive power of the language. Indeed, if terms of the form $f(n, T_2, \dots, T_n)$ appear in right-hand sides of equations and $n \in \mathbb{N}_\delta$, we can replace $f(n, T_2, \dots, T_n)$ with $f(T, T_2, \dots, T_n)$, where

$$T \equiv \text{zero} \text{ iff } n = 0, T \equiv \text{succ } m \text{ iff } n = m + 1.$$

2.1 More on Canonical Systems of Equations

A function definition can be expressed by a canonical set of equations in an extended λ -calculus: let $\Sigma = \Sigma_0 \cup \Sigma_1$ where

$$\Sigma_0 = \{\text{zero}, \text{succ}, \text{nil}, \text{cons}\}, \Sigma_1 = \{\text{Fac}, \text{Map}\};$$

the following declaration is indeed a canonical set of equations in $\Lambda(\Sigma_0 \cup \Sigma_1)$:

$$\begin{aligned} \text{Fac}(\text{zero}) &= 1; & \text{Fac}(\text{succ}(x)) &= *(+1x)(\text{Fac}(x)); \\ \text{Map}(\text{nil}, f) &= \text{nil}; & \text{Map}(\text{cons}(x, L), f) &= \text{cons}(fx, \text{Map}(L, f)). \end{aligned}$$

The given set of equations is clearly safe but not complete.

Any pattern of recursion can be manipulated in such a way to be expressed by a canonical set of equations in the extended λ -calculus; as an example, the Ackermann function can be defined by the following set of equations:

$$\begin{aligned} \text{Ack}(\text{zero}, y) &= +1y; \\ \text{Ack}(\text{succ}(x), \text{zero}) &= \text{Ack}(x, 1); \\ \text{Ack}(\text{succ}(x), \text{succ}(y)) &= \text{Ack}(x, \text{Ack}(+1x, y)). \end{aligned}$$

The above system is not canonical since the last two equations do not have the shape (2), but it is reduced to a canonical system (more precisely, to a safe one) by enlarging the signature with the new function symbol f as follows:

$$\begin{aligned} \text{Ack}(\text{zero}, y) &= +1y; & \text{Ack}(\text{succ}(x), y) &= \text{Ack}(x, f(y, x)); \\ f(\text{zero}, x) &= 1 & f(\text{succ}(z), x) &= \text{Ack}(+1x, z). \end{aligned}$$

To be more general, but still restricting our attention to integer functions let $\Sigma = \Sigma_0 \cup \Sigma_1$, where $\Sigma_0 = \{\text{zero}, \text{succ}\}$, $\Sigma_1 = \{F\}$; a double integer recursion scheme can be presented by means of the following set of equations, where $h_0, \dots, h_3 \in \Lambda(\Sigma)$:

$$\begin{aligned} F(\text{zero}, \text{zero}) &= h_0; & F(\text{zero}, \text{succ}(y)) &= h_1; \\ F(\text{succ}(x), \text{zero}) &= h_2; & F(\text{succ}(x), \text{succ}(y)) &= h_3. \end{aligned} \tag{3}$$

The scheme (3) is not a canonical set of equations in $\Lambda(\Sigma)$, but it can be easily reduced to a mutual recursion scheme by enlarging the signature Σ with two extra function symbols; the resulting set of equations is a canonical one:

$$\begin{aligned} F(zero, z) &= F'(z); & F(succ(x), z) &= F''(z, x); \\ F'(zero) &= h_0; & F'(succ(y)) &= h_1; \\ F''(zero, x) &= h_2; & F''(succ(y), x) &= h_3. \end{aligned} \tag{4}$$

Such reduction is easily proved to be correct, just considering the four possible cases in (3) and verifying they are well defined by (4).

Similarly every partial recursive function can be defined by a canonical system (it is enough to verify closure under minimization, as in [29]). Moreover the correspondence between recursive schemes and canonical systems can be extended to functionals defined over arbitrary algebraic data structures in a straightforward way, as our example of the functional *Map*. Our choice of canonical sets of equations has been made to automatize the execution of *simple pattern matching*, a paradigm used by compilers for functional languages and rewriting systems (see e.g. [25, 2]).

3 Solving equations inside λ -calculus

We have introduced a language which is based on definitions of recursive functions; clearly, a function has an implicitly infinite character.

Historically, solutions of systems of recursive equations are based on the use of fixed point combinators (or similar tools [21, 31]) and yield combinators which make the mentioned infinite character explicit. Such solutions encode all possible unfoldings of functions. Any single datum establishes how many unfoldings of the obtained combinator will be executed, but this, being itself the engine of recursion, is an infinite object from the standpoint of reduction in that it does not have a normal form. To use a slogan, we can say that in this setting, functionals (the fixed point combinator, in particular) are diverging objects which, when applied to data, may “incidentally” converge.

Among the consequences of this, implementations must resort to compute *weak head normal forms* instead of normal forms. Abramsky and Ong introduced the *lazy* λ -calculus [1] to match such implementations.

A different approach is what we propose in this paper, aiming to solve recursive equations without making their infinite character explicit. In our approach, programs are, whenever possible, normal forms; reduction is started only when they receive some input; data become the engine of recursion in that every datum encodes the number of unfoldings it will cause to be executed by any program; hence the infinite characteristics of recursive functions are distributed over an infinity of finite objects. On the other hand, the solution of a set of recursive equations is a combinator encoding exactly the information specified by the equations. To substantiate these ideas we will exhibit an always diverging function represented by a normal form, so that, rephrasing the slogan above, we

can say that in our setting functionals are normal forms which, when applied to data, may “incidentally” diverge.

A *representation* of the signature Σ in the λ -calculus is a function $\phi : \Sigma \rightarrow \Lambda$. Any such representation ϕ induces a map $(\cdot)^\phi : \Lambda(\Sigma) \rightarrow \Lambda$ in the obvious way, namely

$$\text{for any atomic symbol } a, \quad a^\phi = \begin{cases} \phi(a) & \text{if } a \in \Sigma \\ a & \text{otherwise,} \end{cases}$$

$$(\lambda x.M)^\phi = \lambda x.M^\phi,$$

$$(MN)^\phi = M^\phi N^\phi,$$

$$f(M_1, \dots, M_n)^\phi = f^\phi M_1^\phi \dots M_n^\phi.$$

Definition 3. Let $\mathcal{E} = \{a_i = b_i | i \in J\}$ be a set of equations between extended lambda terms $a_i, b_i \in \Lambda(\Sigma)$.

1. We say that a representation ϕ *satisfies* (or *solves*) \mathcal{E} if for each equation $a_i = b_i$ in \mathcal{E} we have $a_i^\phi =_\beta b_i^\phi$. If there exists a representation ϕ which satisfies \mathcal{E} we say that \mathcal{E} can be *interpreted* (or *represented* or *solved*) inside λ -calculus and that ϕ is a *solution* for \mathcal{E} .
2. A solution ϕ for \mathcal{E} is called a *normal solution* if, for all h in Σ , $\phi(h)$ is a β -normal form.

Theorem 4 (Interpretation Theorem).

Let $\Lambda(\Sigma)$ be an extended λ -calculus; then every safe set of equations \mathcal{E} has a normal solution $\phi : \Sigma \rightarrow \Lambda$ inside λ -calculus. Furthermore we can choose ϕ so that the restriction $\phi|_{\Sigma_0}$ depends only on Σ_0 and not on \mathcal{E} , namely there is a fixed representation of the constructors.

Proof. Let $\Sigma_0 = \{c_1, c_2, \dots, c_r\}$.

For $1 \leq j \leq r$, we define $\vartheta = \phi|_{\Sigma_0} : \Sigma_0 \rightarrow \Lambda$:

$$\vartheta(c_j) = \lambda x_1 \dots x_m. e. e \mathbf{U}_j^r x_1 \dots x_m, \tag{5}$$

where m is the arity of c_j and $\mathbf{U}_j^r \equiv \lambda x_1 \dots x_r. x_j$.

It remains to define $\zeta = \phi|_{\Sigma_1} : \Sigma_1 \rightarrow \Lambda$, namely the representation of programs. Without loss of generality we can assume that \mathcal{E} is complete (otherwise adjoin more equations to make it complete).

Let $\Sigma_1 = \{f_1, \dots, f_k\}$. Consider $k \times r$ lambda terms $t_{i,j}$, $1 \leq i \leq k$, $1 \leq j \leq r$ to be defined later. Recall the definition of *Church n-tuple*:

$$\langle M_1, \dots, M_n \rangle \equiv \lambda x. x M_1 \dots M_n.$$

For $1 \leq i \leq k$, let $t_i \equiv \langle t_{i,1}, \dots, t_{i,r} \rangle$ and define

$$\zeta(f_i) \equiv \langle t_i, t_1, t_2, \dots, t_k \rangle.$$

Thus $\zeta(f_i)$ is a Church $k + 1$ -tuple of Church r -tuples of terms. The lambda terms $t_{i,j}$ are chosen in the only natural way which makes ζ a solution of the canonical system of equations \mathcal{E} . More precisely consider the equation

$$f_i(c_j(x_1, \dots, x_m), y_1, \dots, y_n) = b_{i,j}$$

belonging to \mathcal{E} ($b_{i,j} \in A(\Sigma)$). After applying $\phi = \vartheta \circ \zeta$ the equation becomes

$$\langle t_i, t_1, \dots, t_k \rangle (c_j^\phi x_1 \dots x_m) y_1 \dots y_n = b_{i,j}^\phi.$$

By definition of Church tuple, this simplifies to

$$c_j^\phi x_1 \dots x_m t_i t_1 \dots t_k y_1 \dots y_n = b_{i,j}^\phi.$$

Recalling the definition of c_j^ϕ we have

$$c_j^\phi x_1 \dots x_m t_i = t_i \mathbf{U}_j^r x_1 \dots x_m = t_{i,j} x_1 \dots x_m.$$

Hence the equation becomes

$$t_{i,j} x_1 \dots x_m t_1 \dots t_k y_1 \dots y_n = b_{i,j}^\phi.$$

We can now solve this equation for $t_{i,j}$ by replacing on both sides all the occurrences of t_1, \dots, t_k by fresh variables v_1, \dots, v_k and abstracting with respect to all variables present in left-hand-side. More precisely define:

$$t_{i,j} \equiv \lambda x_1 \dots x_m v_1 \dots v_k y_1 \dots y_n. (b_{i,j}^\phi)^\psi$$

where $\psi : \Sigma_1 \rightarrow A$ is defined by

$$\psi(f_i) = \langle v_i, v_1, \dots, v_k \rangle. \quad (6)$$

Note that, for any $V \in A(\Sigma)$, $V^\zeta = V^\psi [t_h/v_h]_{1 \leq h \leq k}$.

With this definition

$$\begin{aligned} t_{i,j} x_1 \dots x_m t_1 \dots t_k y_1 \dots y_n &\rightarrow (b_{i,j}^\phi)^\psi [t_h/v_h]_{1 \leq h \leq k} \\ &= b_{i,j}^{\vartheta \circ \zeta} = b_{i,j}^\phi \end{aligned}$$

and all the equations will be satisfied.

We now prove that the given technique yields normal solutions for safe systems of equations.

Let \mathcal{E} be safe and let $\vartheta : \Sigma_0 \rightarrow A$ and $\psi : \Sigma_1 \rightarrow A$ be as in (5) and (6), respectively.

For any equation $a = b \in \mathcal{E}$, we first prove by induction on the number of occurrences of constructors in b that b^ϑ is a normal form. Indeed, if constructors do not appear in b , then $b^\vartheta = b$, a normal form (by definition 2.i); if $c_j(X_1, \dots, X_m)$ occurs in b , for some $X_1, \dots, X_m \in A(\Sigma)$, then

$$c_j(X_1, \dots, X_m)^\vartheta = \lambda e. e \mathbf{U}_j^r X_1^\vartheta \dots X_m^\vartheta$$

which is, by inductive hypothesis, a normal form and, by 2.iii, does not create any new redex.

It comes out from 2.ii that for any equation $a = b \in \mathcal{E}$, b^ϑ is such that $\forall f \in \Sigma_1. \forall T_1, \dots, T_n \in A(\Sigma)$ if $f(T_1 \dots T_n)$ occurs in b^ϑ then T_1 either has no initial abstractions or it has the shape $\lambda e. e \mathbf{U}_j^r X_1 \dots X_n$, for some $X_1, \dots, X_n \in$

$\Lambda(\Sigma)$. Now, if programs do not appear in b^ϑ , then $(b^\vartheta)^\psi = b^\vartheta$, a normal form; if $f_i(T_1, \dots, T_n)$ occurs in b^ϑ , for some $T_1, \dots, T_n \in \Lambda(\Sigma)$, then

$$f_i(T_1, \dots, T_n)^\psi = T_1^\psi v_i v_1 \dots v_k T_2^\psi \dots T_n^\psi$$

which reduces in at most one step to a head normal form without any initial abstraction. It follows by induction that $(b^\vartheta)^\psi$ reduces to a normal form, so that, applying to such normal form the construction in the first part of the proof of the theorem, we obtain a normal solution for \mathcal{E} .

Example 1. Given $\Sigma = \Sigma_0 \cup \Sigma_1$ where

$$\Sigma_0 = \{\text{zero}, \text{succ}, \text{nil}, \text{cons}\} \text{ and } \Sigma_1 = \{\text{Fac}, \text{Map}\},$$

let \mathcal{E} be the following set of equations:

$$\begin{aligned} \text{Fac(zero)} &= 1; \\ \text{Fac(succ}(x)) &= *(+1x)(\text{Fac}(x)); \\ \text{Map(nil, f)} &= \text{nil}; \\ \text{Map(cons}(x, L), f) &= \text{cons}(fx, \text{Map}(L, f)). \end{aligned}$$

We can complete \mathcal{E} adding the equations

$$\begin{aligned} \text{Fac(nil)} &= \text{Type_err}_1; \\ \text{Fac(cons}(x, L)) &= \text{Type_err}_1; \\ \text{Map(zero, f)} &= \text{Type_err}_2; \\ \text{Map(succ}(x), f) &= \text{Type_err}_2, \end{aligned}$$

where Type_err_1 and Type_err_2 are two variables.

If we assume

$$\begin{aligned} \phi(\text{zero}) &= \lambda e. e \mathbf{U}_1^4, \quad \phi(\text{succ}) = \lambda x e. e \mathbf{U}_2^4 x, \\ \phi(\text{nil}) &= \lambda e. e \mathbf{U}_3^4, \quad \phi(\text{cons}) = \lambda x L e. e \mathbf{U}_4^4 x L, \\ \phi(\text{Fac}) &= \langle t_1, t_1, t_2 \rangle, \quad \text{where } t_1 = \langle t_{1,1}, \dots, t_{1,4} \rangle \\ \phi(\text{Map}) &= \langle t_2, t_1, t_2 \rangle, \quad \text{where } t_2 = \langle t_{2,1}, \dots, t_{2,4} \rangle \end{aligned}$$

and we consider the derived set of equations, we obtain

$$\begin{aligned} t_1 \mathbf{U}_1^4 t_1 t_2 &= 1; \\ t_1 \mathbf{U}_2^4 x t_1 t_2 &= *(+1x)(x t_1 t_1 t_2); \\ t_1 \mathbf{U}_3^4 t_1 t_2 &= t_1 \mathbf{U}_4^4 x L t_1 t_2 = \text{Type_err}_1; \\ t_2 \mathbf{U}_1^4 t_1 t_2 f &= t_2 \mathbf{U}_2^4 x t_1 t_2 f = \text{Type_err}_2; \\ t_2 \mathbf{U}_3^4 t_1 t_2 f &= \phi(\text{nil}); \\ t_2 \mathbf{U}_4^4 x L t_1 t_2 f &= \phi(\text{cons})(fx)(L t_2 t_1 t_2 f); \end{aligned}$$

hence, we have

$$\begin{aligned}
 t_{1,1} t_1 t_2 &= 1; \\
 t_{1,2} x t_1 t_2 &= *(+1x)(x t_1 t_1 t_2); \\
 t_{1,3} t_1 t_2 &= t_{1,4} x L t_1 t_2 = Type_err_1; \\
 t_{2,1} t_1 t_2 f &= t_{2,2} x t_1 t_2 f = Type_err_2; \\
 t_{2,3} t_1 t_2 f &= \phi(nil); \\
 t_{2,4} x L t_1 t_2 f &= \phi(cons)(fx)(L t_2 t_1 t_2 f);
 \end{aligned}$$

which is solved taking

$$\begin{aligned}
 t_{1,1} &\equiv \lambda v_1 v_2. 1, \quad t_{1,2} \equiv \lambda x v_1 v_2. * (+1x)(x v_1 v_1 v_2), \\
 t_{1,3} &\equiv \lambda v_1 v_2. Type_err_1, \quad t_{1,4} \equiv \lambda x_1 x_2 v_1 v_2. Type_err_1, \\
 t_{2,1} &\equiv \lambda v_1 v_2 f. Type_err_2, \quad t_{2,2} \equiv \lambda x v_1 v_2 f. Type_err_2, \\
 t_{2,3} &\equiv \lambda v_1 v_2 f. \phi(nil), \\
 t_{2,4} &\equiv \lambda x_1 x_2 v_1 v_2 f. \phi(cons)(fx_1)(x_2 v_2 v_1 v_2 f).
 \end{aligned}$$

It follows that the representation for *Fac* and *Map* is a Church 3-tuple of Church 4-tuples of normal forms, hence a normal form.

Remark. It has to be noted that we obtain normal solutions also for definitions of intrinsically diverging programs. Given $\Sigma = \Sigma_0 \cup \Sigma_1$ where

$$\Sigma_0 = \{zero, succ\} \text{ and } \Sigma_1 = \{F\},$$

let \mathcal{E} be the following set of equations:

$$\begin{aligned}
 F(zero) &= F(zero); \\
 F(succ(x)) &= F(succ(x)),
 \end{aligned} \tag{7}$$

First observe that such system is safe, in spite of the fact that it defines an always diverging function. If we assume

$$\begin{aligned}
 \phi(zero) &= \lambda e.e \mathbf{U}_1^2, \quad \phi(succ) = \lambda x e.e \mathbf{U}_2^2 x, \\
 \phi(F) &= \langle t_1, t_1 \rangle, \quad \text{where } t_1 = \langle t_{1,1}, t_{1,2} \rangle
 \end{aligned}$$

and we consider the derived set of equations, we obtain

$$\begin{aligned}
 t_1 \mathbf{U}_1^2 t_1 &= t_1 \mathbf{U}_1^2 t_1; \\
 t_1 \mathbf{U}_2^2 x t_1 &= t_1 \mathbf{U}_2^2 x t_1;
 \end{aligned}$$

hence, we have

$$\begin{aligned}
 t_{1,1} t_1 &= t_1 \mathbf{U}_1^2 t_1; \\
 t_{1,2} x t_1 &= t_1 \mathbf{U}_2^2 x t_1;
 \end{aligned}$$

which is solved taking

$$t_{1,1} \equiv \lambda v_1. v_1 \mathbf{U}_1^2 v_1, \quad t_{1,2} \equiv \lambda x v_1. v_1 \mathbf{U}_2^2 x v_1.$$

It follows that the representation for *F* is a Church 2-tuple of Church 2-tuples of normal forms, hence a normal form.

3.1 A double citizenship for data

The Interpretation Theorem 4 enables to obtain normal solutions for safe systems of equations.

The key idea allowing such result is that data are considered as functionals, interpreting them in λ -calculus. Now, a natural question arises: are we willing to fully represent data structures in λ -calculus? The answer is surely negative, for we want to preserve the structure of data during computation. Roughly speaking, we would like to give data a double citizenship: they should act as functionals during function application, otherwise preserving their original status.

The Interpretation Theorem itself hints to a satisfactory solution to this problem: being the representation of data constructors fixed, we will consider them as predefined combinators, i.e. extra constants to be included in Λ ; furthermore, we will consider their functional behaviour to be defined by *weak reduction rules*, in such a way that a constructor with arity m needs (by the weak version of (5)) at least $m + 1$ arguments to be reduced. Assuming then $\{c_1, \dots, c_r\} \in \Lambda$ with

$$c_j X_1 \dots X_m E \triangleright E U_j^r X_1 \dots X_m,$$

where ' \triangleright ' denotes weak reduction, it turns out that all definitions and results obtained in the previous subsections still hold taking now $\Sigma \equiv \Sigma_1$, since $\Sigma_0 = \emptyset$, being the constructors considered constants in Λ .

Finally, the coexistence of different representations for natural numbers is ruled by the following notion of reduction (χ -reduction) which allows rewriting δ -integers when these appear in functional position in an application:

$$\frac{0 \in \mathbb{N}_\delta \quad T \in \Lambda}{0 T \xrightarrow{\chi} \text{zero}^\phi T}, \quad \frac{n = m+1 \in \mathbb{N}_\delta \quad T \in \Lambda}{n T \xrightarrow{\chi} \text{succ}^\phi m T}.$$

Example 2. (Ex.1 continued)

To give the example of a computation, let $\phi(Map)$ be as in example 1. We have e.g. (superscript ϕ is sometimes omitted below)

$$\begin{aligned} & (Map(\text{cons}(1, \text{cons}(2, \text{nil})), f))^\phi \\ = & \text{cons } 1(\text{cons } 2 \text{ nil})t_2 t_1 t_2 f \\ \triangleright & t_2 U_4^4 1(\text{cons } 2 \text{ nil})t_1 t_2 f \\ \longrightarrow_\beta & t_2, 4 1(\text{cons } 2 \text{ nil})t_1 t_2 f \\ \longrightarrow_\beta & \text{cons}(f1)(\text{cons } 2 \text{ nil} t_2 t_1 t_2 f) \\ \triangleright & \dots \longrightarrow_\beta \text{cons}(f1)(\text{cons}(f2)(\text{nil} t_2 t_1 t_2 f)) \\ \triangleright & \text{cons}(f1)(\text{cons}(f2)(t_2 U_3^4 t_1 t_2 f)) \\ \longrightarrow_\beta & \text{cons}(f1)(\text{cons}(f2)\text{nil}), \text{ a } \triangleright\text{-normal form.} \end{aligned}$$

On the other hand, we have:

$$\begin{aligned} & (Map(7, f))^\phi \\ = & 7 t_2 t_1 t_2 f \xrightarrow{\chi} \text{succ } 6 t_2 t_1 t_2 f \\ \triangleright & t_2 U_2^4 6 t_1 t_2 f \longrightarrow_\beta t_2, 2 6 t_1 t_2 f \\ \longrightarrow_\beta & \text{Type_err}_2. \end{aligned}$$

Summarizing, \triangleright -normal forms are important tools to recognize algebraic objects as results of computations. This solves a problem mentioned in [5], in the context of self-interpretation of λ -calculus.

4 Further Properties: some examples

This section is devoted to the illustration of possible applications in functional programming of the theoretical issues just presented.

The following examples refer to recursive definitions (*D*) of function(al)s, execution commands (*E*) and results (*R*) of computations based on the implementation of the methods described in this paper, called *CuCh-machine*. This is an acronym for Curry and Church, first introduced in [12, 13] to describe a machine simultaneously accepting combinators and λ -terms and reducing them to normal form. Further properties not exemplified below are the allowance of free variables, and the use of lazy data structures, in the style of [18], implemented by normal order reduction (to be compared with [20]).

4.1 Currying

```
(D) ACK zero f := f 1;
(D) ACK (succ m)f := f (ACK m f);
(D) ack zero x := + 1 x;
(D) ack (succ n) x := ACK x (ack n);
```

Curried ack

```
(E) ack3 := ack 3;
(R)  $\lambda x_0 . \text{ACK } x_0 (\lambda x_1 . \text{ACK } x_1 (\lambda x_2 . \text{ACK } x_2 (\lambda x_3 . + 1 x_3)))$  (3 beta)
(E) ack34 := ack3 4;
(R) 125 (15520 beta)
(E) ack35 := ack3 5;
(R) 253 (63780 beta)
```

Non curried ack

```
(E) r := ack 3 4;
(R) 125 (15640 beta)
(E) s := ack 3 5;
(R) 253 (64027 beta)
```

4.2 Iterative Functions

(see [19])

```
(D) map :=  $\lambda x_0 x_1 x_2 x_3 . x_1 (\lambda x_4 x_5 . x_2 (x_0 x_4) x_5) x_3;$ 
(E) mapmap :=  $\lambda x . \text{map } f(\text{map } g x);$ 
(R)  $\lambda x_0 x_1 x_2 . x_0 (\lambda x_3 x_4 . x_1 (f (g x_3)) x_4) x_2$ 
(E) comp :=  $\lambda x . \text{map } (B f g) x;$ 
(R)  $\lambda x_0 x_1 x_2 . x_0 (\lambda x_3 x_4 . x_1 (f (g x_3)) x_4) x_2$ 
(D) foldr nil a b := b;
(D) foldr (cons x L) a b := a x(foldr L a b);
(D) list := [1,4,5,6,7];
```

```
(E) flist := foldr list;
(R)  $\lambda x_0 x_1 . x_0 \ 1 \ (x_0 \ 4 \ (x_0 \ 5 \ (x_0 \ 6 \ (x_0 \ 7 \ x_1))))$ 
(E) bb := mapmap flist cons nil;
(R) [ f (g 1), f (g 4), f (g 5), f (g 6), f (g 7) ]
```

4.3 Complete Systems

```
(D) mbf zero x := $\lambda y.$ mbf y x;
(D) mbf (succ n) x:= $\lambda y.$ mbf y (+ x(+ 1 n));
(D) mbf nil x:= x;
(D) mbf(cons t l)x:= mbf l (+ t x);
(E) ee:= mbf [1, 3] 0;
(R) 4
(E) xy:= mbf 7 6 4 nil;
(R) 17
```

Acknowledgments

We would like to thank Mariangiola Dezani-Ciancaglini and Ugo de'Liguoro for helpful discussions and suggestions about the topics of this paper. We are grateful to the referees of the preliminary version of the paper for their criticism and suggestions for improving the presentation.

References

1. S. Abramsky, C.-H.L. Ong, *Full Abstraction in the Lazy Lambda Calculus*, Technical Report 259, Cambridge University Computer Laboratory, 1992, 105 pp. To appear in *Info. and Comp.*
2. L.Augustsson and T.Johnsson, *The Chalmers Lazy-ML Compiler*, The Computer Journal, vol. 32, no. 2, April 1989.
3. J.Backus, *Can programming be liberated from vonNeumann style? A functional style and its algebra of programs*, ACM Comm.,1978, vol.21, no. 8, pp. 613-641.
4. H.P.Barendregt, *The type free lambda-calculus*, in: Handbook of Mathematical Logic, Barwise (ed.), North Holland, 1981, pp.1092-1132.
5. A.Berarducci and C.Böhm, *A self-interpreter of lambda calculus having a normal form*, 6th Workshop CSL '92, San Miniato, Italy, September-October 1992, eds E. Börger et al., Springer Verlag, Berlin (LNCS 702), pp. 85-99.
6. C.Böhm, *Combinatory foundation of functional programming*, in 1982 ACM Symposium on Lisp and functional programming, 1982, Pittsburgh, Pen., pp.29-36.
7. C. Böhm, *Reducing Recursion to Iteration by Algebraic Extension* in: ESOP 86, (LNCS 213), p.111-118, 1986.
8. C. Böhm, *Reducing Recursion to Iteration by means of Pairs and N-tuples*, in: Foundations of Logic and Functional Programming, LNCS 306, p.58-66, 1988.
9. C.Böhm, *Functional Programming and Combinatory Algebras*, MFCS, Carlsbad, August-September 1988, eds M. P. Chytil et al., Springer Verlag, Berlin (LNCS 324), pp. 14-26.
10. C.Böhm, *Subduing Self-Application*, ICALP '89, Stresa, July 11-15 1989, eds G. Ausiello et al., Springer Verlag, Berlin (LNCS 372), pp. 108-122.

11. C.Böhm and A.Berarducci, *Automatic Synthesis of Typed λ -Programs on Term Algebras*, Theoretical Computer Science 39, pp. 135–154, 1985.
12. C.Böhm and M.Dezani-Ciancaglini, *A CUCH-machine: the automatic treatment of bound variables*, International Journal of Computer and Information Sciences, vol. 1, no. 2, pp. 171–191, June 1972.
13. C.Böhm and M.Dezani-Ciancaglini, *Notes on “A CUCH-machine: the automatic treatment of bound variables”*, International Journal of Computer and Information Sciences, vol. 2, no. 2, pp. 157–160, June 1973.
14. C.Böhm and M.Dezani-Ciancaglini, *Combinatorial problems, combinator equations and normal forms*, in: Loeckx (ed.) Automata, Languages and Programming 2th. Colloquium, LNCS 14, 1974, pp.185-199.
15. C.Böhm and M.Dezani-Ciancaglini, *λ -terms as total or partial functions on normal forms*, in: λ -Calculus an computer science theory Böhm (ed.), LNCS 37, Springer, 1975, pp.96-121.
16. A.Church, *The calculi of lambda-conversion*, Princeton Univ.Press, 1941.
17. H.B.Curry, *Combinatory Logic*, Vol I, North Holland, Amsterdam, 1958.
18. D.P.Friedman and D.S.Wise, *Cons should not evaluate its arguments*, Proc.3rd International Colloquium on Automata, Languages and Programming, Edinburgh, 1976, pp.257–284.
19. A.Gill, J.Launchbury and S.L.Peyton-Jones, *A Short Cut to Deforestation*, Functional Programming and Computer Architecture, 1993.
20. J.Hughes, *Why Functional Programming Matters*, The Computer Journal, special issue on Lazy Functional Programming, vol. 32, no. 2, April 1989.
21. J.Hughes, *Supercombinators: a new implementation method for applicative languages*, Symp. on LISP and Functional Programming, ACM, 1982.
22. S.C.Kleene, *λ -definability and recursiveness*, Duke Math.J. 2, pp.340-353.
23. D.E.Knuth, *The Art of Computer Programming*, Vol. 1/Fundamental Algorithms, Addison-Wesley, 1973.
24. M. Parigot. *Programming with proofs: a second order type theory*, ESOP'88, LNCS 300, pp. 145-159.
25. S.L.Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice-Hall, 1986.
26. H.Schwichtenberg, *Einige Anwendungen von unendlichen Termen und Wertfunktionalen*, Habilitationsschrift, Münster, 67 pp., 1973.
27. J.R.Shoenfield, *Mathematical Logic*, Addison Wesley, 1967.
28. H.R. Strong, *Algebraically Generalized Recursive Function Theory*, IBM J.Res. Develop.12 (1968), pp.465-475.
29. E.Tronci, *Equational programming in lambda-calculus*, Proc.of LICS'91, IEEE Comp.Soc., 1991.
30. A.Turing, *On computable numbers with an application to the Entscheidungsproblem*, Proc.London Math.Soc. 42, pp.230-265.
31. D.A.Turner, *A new implementation technique for applicative languages*, Software practice and experience, no. 9, 1979.
32. E.G.Wagner, *Uniformly Reflexive Structures: An Axiomatic Approach to Computability*, Information Sci.1 (1969), pp.343-362.

Simulation of SOS Definitions with Term Rewriting Systems

Karl-Heinz Buth*

Institut für Informatik und Praktische Mathematik
Christian-Albrechts-Universität Kiel
Preusserstr. 1–9, D–24105 Kiel, Germany
e-mail: khb@informatik.uni-kiel.d400.de

Abstract. Reasoning about programming language semantics with an automated proof tool requires that the semantics definition be stated in a formalism that is suitable for the tool. This paper presents a method to transform a structured operational semantics (SOS) definition \mathcal{S} , given by a special form of deduction system, into a term rewriting system \mathcal{R} . This system \mathcal{R} simulates \mathcal{S} very closely in that sense that the sets of possible configuration sequences are essentially the same. Since only standard unconditional rewrite rules are used, every theorem prover based on rewriting can be employed to implement this kind of semantics definitions, and so to reason about them.

1 Introduction

A common way to describe the semantics of a programming language is to give an operational definition. This means that an abstract machine is introduced, and that the elements of the language are explained in terms of the machine instructions. If we want to reason about such a definition with the help of a proof support system, we have to express the explanations within the tool's formalism.

In this paper, we demonstrate a way how this can be done for structured operational semantics (SOS) definitions in the sense of Plotkin (cf. [21]), and proof tools based on term rewriting. An SOS definition is given by means of a transition system, where the transition relation is presented in form of a set of deduction rules. These rules cannot be used directly as rewrite rules since the conditions on the variables that occur in rewrite rules are more restrictive than those required for deduction rules. Our way to solve this problem is to use a limited subset of λ -calculus to model the rules. This subset can be expressed completely by rewrite rules; thus no new formalism is needed. We can prove that our method leads to a very close simulation of the original transition system. Since only simple rewrite rules are used, the method can be implemented with every proof tool that supports term rewriting.

* Partially supported by Esprit BRA projects 3104 “ProCoS” and 7071 “ProCoS II” and Deutsche Forschungsgemeinschaft, grants La 426/12-1 and La 426/12-2.

An application of this approach is presented in [7], where the Larch Prover [11] is used as a proof assistant in an equivalence proof for different semantics definitions. The motivation for the work emerged from verification work in the ESPRIT BRA project ProCoS (*Provably Correct Systems*, cf. [4]). Several of the ProCoS project languages have been defined operationally, and therefore there is a need to reason about SOS definitions.

We start in section 2 with a short account of term rewriting, followed in section 3 by an introduction to SOS definitions. Section 4 presents the transformation of deduction into rewrite rules. A summary of the simulation properties of the derived rewriting system is given in section 5. The proofs for the results can be found in the full version of this paper ([6]). Section 6 describes in which way the simulating rewrite system can be applied.

2 Term rewriting controlled by contexts

The approach to term rewriting we use in our simulation is a typed one, based on a many-sorted logic (for details, cf. [9]). We only employ unconditional rewrite rules; so any standard tool can be used for the implementation of the ideas.

A **term rewriting system** (TRS for short) is a finite set of rules $\lambda \rightarrow \rho$, where λ and ρ are terms of the same type, and ρ does not contain *extra variables* w. r. t. λ , i. e. all variables in ρ are also in λ . An ***f*-term** is a term whose outermost operator is *f*. An ***f*-rule** is a rule $\lambda \rightarrow \rho$ where the left-hand side λ is an *f*-term. For a signature Σ and a set of variables V , $T(\Sigma, V)$ denotes the set of all terms built over Σ and V . For $n \in \mathbb{N}_0$, we define $[n] =_{df} \{1, \dots, n\}$.

Our simulation relies on terms of the form $T \equiv \text{let } x = e_1 \text{ in } e_2$, since this is a convenient way to give names to intermediate results: x is a name for the result of “evaluating” e_1 . Now T is just another notation for $(\lambda x. e_2)e_1$, and so we have started using concepts of λ -calculus. But we need not merge rewriting and β -reduction as it is done in e. g. [10, 18]; since we only have λ -abstractions that are directly applied to non-functional arguments, we can use standard rewrite systems to evaluate β -reductions. This method, presented e. g. in [1], is based on the replacement of bound variables by de Bruijn indices ([5]). For details, cf. [6]; in the following, we will only use the **let** form to write such terms.

In order to obtain a correct simulation, we need to control the rewriting process in two ways (for the examples, assume that *f* is a unary operator):

- We want to be able to perform just one top-level rewriting of an *f*-term t without restricting the rewriting of subterms. This cannot be done by ordinary methods like restricting the length of rewriting sequences to 1.
- We want to allow a rule of the form

$$f(t_1) \rightarrow \text{if } b \text{ then } f(t_2) \text{ else } f(t_1)$$

This rule is obviously non-terminating (in the usual sense), but we want to be able to apply it without successive selections of the else part. This means that when an instance of *b* is known to reduce to **false**, the corresponding instance of this rule shall be “de-activated”.

For the first wish, we need to have some kind of counter, and for the second a way of “switching a rule off” (and on again, of course). This is the motivation for the following definition.

Definition 1. Let $l_1 \rightarrow r_1, \dots, l_n \rightarrow r_n$ be all the f -rules in our rewrite system. Then a **context for f** is an element of $\{0, 1, *\} \times \{\text{on}, \text{off}\}^n$. For $a \in \{0, 1, *\}$ and $s_1, \dots, s_n \in \{\text{on}, \text{off}\}$, the context $\langle a, s_1, \dots, s_n \rangle$ is called an **a -context**.

Contexts can be used for our purposes since they contain a counter component (0, 1 or *) and a switch for each f -rule. Instead of rewriting terms $f(t)$, we now rewrite terms $f(t) @ \langle a, s_1, \dots, s_n \rangle$ where $@$ is the special context application operator, $a \in \{0, 1, *\}$ and $s_1, \dots, s_n \in \{\text{on}, \text{off}\}$. The intended interpretation for a is:

- no more top-level rewriting steps, if $a = 0$,
- at most one top-level rewriting step, if $a = 1$,
- no limit on the number of top-level rewriting steps, if $a = *$.

The interpretation for the s_i is that application of the i -th rule is allowed if $s_i = \text{on}$ and disallowed otherwise.

But of course it does not suffice to modify the term that shall be rewritten. We also have to build the control mechanism into the f -rules $l_1 \rightarrow r_1, \dots, l_n \rightarrow r_n$. This is done by supplying all f -terms in all l_i and r_j with appropriate contexts: the i -th rule

$$f(t_1) \rightarrow \text{if } b \text{ then } f(t_2) \text{ else } f(t_1)$$

becomes (assume that b, t_1 and t_2 do not contain f -terms)

$$f(t_1) @ \langle 1, s_1, \dots, s_{i-1}, \text{on}, s_{i+1}, \dots, s_n \rangle \rightarrow$$

$$\text{if } b \text{ then } f(t_2) @ \langle 0, \text{on}, \dots, \text{on} \rangle$$

$$\text{else } f(t_1) @ \langle 1, s_1, \dots, s_{i-1}, \text{off}, s_{i+1}, \dots, s_n \rangle$$

In the **then** case, all rules are switched on, since the rule has been successfully applied. Switching rule i off in the **else** case is the “de-activation” that was mentioned above.

Usually, the introduction of contexts restricts the rewriting relation of a given system \mathcal{R} . Certain unwanted rewriting sequences are thrown out, but no additional sequences become possible. This is because every rewriting step in the system with contexts corresponds to a step in the original system; the terms themselves are not changed.

In order not to complicate the control of rewriting, we demand that f -rules must not have nested f -terms on their left-hand sides. This means that the connection between a rule and the corresponding switch position in contexts is easy to maintain.

3 Structured operational semantics

The operational definition of the semantics of a programming language L is accomplished by first defining an abstract machine M and then interpreting the constructs of L by means of the machine instructions of M . In Plotkin’s approach, M is given in the form of a transition system:

Definition 2. A **transition system** is a triple (Γ, T, \rightarrow) , where Γ is the set of **configurations**, $T \subseteq \Gamma$ is the set of **terminal configurations**, and $\rightarrow \subseteq \Gamma \times \Gamma$ is the **transition relation** satisfying $(T \times \Gamma) \cap \rightarrow = \emptyset$.

In order to ease the modelling of interactions with the environment, transitions are usually *labelled* with *actions*: $\gamma_1 \xrightarrow{a} \gamma_2$ then means that the step from configuration γ_1 to γ_2 is taken while performing some action a together with the program's environment. Typical actions of that kind are communication events. There is, however, no greater expressive power in labelled systems. They can be simulated by unlabelled systems whose configurations have an extra component that contains the sequences of labels; therefore, there is no need to consider labelled systems.

The starting point in defining a transition system is the definition of the configurations. So let us assume the two sets Γ and T given. Furthermore, let us assume a signature Σ and a set of variables V such that $T(\Sigma, V)$ contains all the terms that we need to express configurations, contexts, and other mathematical objects we need. Special subsets of $T(\Sigma, V)$ are Γ' and T' , representing schemata for configurations and terminal configurations, respectively. If no confusion can arise, we will identify configurations and their term representations.

The transition relation \rightarrow is now defined by means of a special kind of deduction system:

Definition 3. An **SOS deduction system** for Γ and T consists of the term sets $T(\Sigma, V)$, Γ' and T' , and inference rule schemata of the following kind:

$$\frac{\vdash \bigwedge_{i=1}^p b_i \wedge \bigwedge_{j=1}^n \gamma_j \xrightarrow{L_j} \gamma'_j \wedge \bigwedge_{k=1}^q B_k}{\vdash \bar{\gamma} \rightarrow \bar{\gamma}'}$$

The b_i are basic predicate terms restricting the input variables in $\bar{\gamma}$ ("preconditions"), and the B_j are predicate terms restricting the output variables not in $\bar{\gamma}$ ("postconditions"). In transitions, extra variables w. r. t. $\bar{\gamma}$ must only occur on the right-hand side; all variables in $\bar{\gamma}'$ must also occur in some other configuration term. L_j may be 1, denoting one-step transition, or *, denoting an arbitrary number of steps.² In the latter case, γ'_j must be terminal, i. e. a normal form.

The semantics of this kind of inference rules is as usual: An instance of the conclusion is established if the corresponding instance of the hypothesis can be established using the rules of the system. All variables of the rules are implicitly universally quantified.

As a consequence of this definition, the restriction to conjunctions in the hypothesis does not limit the expressive power of the formalism. Any quantifier-free hypothesis can be implemented by first transforming it into disjunctive normal form and then splitting the rule into several rules with the same conclusion, each component of the disjunction forming the hypothesis of a separate rule.

² L_j is *not* an action as they occur in labelled transition systems (see remarks above).

As an additional requirement for SOS deduction systems we demand that the rules do not permit non-terminating proof attempts. The simplest example for a rule that is forbidden is

$$\frac{\vdash \gamma \rightarrow \gamma'}{\vdash \gamma \rightarrow \gamma'}$$

It has the form of definition 3, but it cannot be used for proving any transition. A way to exclude such unpleasant behaviour is to demand that all transitions in the premise of a rule be smaller than the transition in the conclusion w. r. t. some well-founded ordering. The existence of such an ordering is sufficient to prevent non-terminating proof attempts.³

4 An example for transformation

In this chapter, we will informally describe how we can transform SOS deduction rules into term rewriting rules and why some other seemingly “obvious” ways do not work. The exact definition of this transformation can be found in [6].

4.1 The example language definition

As an example (artificial, but not overly simple) let us consider an extract from an imperative language L . In L , there is a syntactic class of statements, denoting state transformations, and the usual operator “;” for sequential composition:

$$\text{Stmt} \ni \text{stmt} ::= \text{stmt}_1; \text{stmt}_2 \mid \dots$$

On the semantic side, we have a set of states Σ that statements can transform. The internal structure of states $\sigma \in \Sigma$ is not important to us. A configuration can either consist of a statement to be executed together with an initial state for this execution, or it can be the final state of an execution:

$$\Gamma_{\text{Stmt}} = \text{Stmt} \times \Sigma \cup T_{\text{Stmt}}$$

$$T_{\text{Stmt}} = \Sigma$$

The execution of a statement list proceeds from left to right. After one computation step, the first statement in a list may have terminated, resulting in a final state, or there may still be a rest of this statement waiting for execution. For these two possibilities, we have the following two inference rule schemata:

$$\frac{\vdash \langle \text{stmt}_1, \sigma \rangle \xrightarrow{\text{Stmt}} \sigma_1}{\vdash \langle \text{stmt}_1; \text{stmt}_2, \sigma \rangle \xrightarrow{\text{Stmt}} \langle \text{stmt}_2, \sigma_1 \rangle} \quad (1)$$

$$\frac{\vdash \langle \text{stmt}_1, \sigma \rangle \xrightarrow{\text{Stmt}} \langle \text{stmt}'_1, \sigma_1 \rangle}{\vdash \langle \text{stmt}_1; \text{stmt}_2, \sigma \rangle \xrightarrow{\text{Stmt}} \langle \text{stmt}'_1; \text{stmt}_2, \sigma_1 \rangle} \quad (2)$$

Rule (1) deals with the case of termination of stmt_1 and (2) with the other case. stmt'_1 is what remains of stmt_1 after one computation step.

³ This is the usual method to prove termination of rewrite systems. There the right-hand side of a rule must be smaller than its left-hand side.

4.2 Transformation into rewrite rules

The simplest possible approach to the problem of transforming a rule

$$\frac{\vdash \text{hypo}}{\vdash \gamma \rightarrow \gamma'}$$

into a rewrite rule is to simulate the rule's semantics ("if *hypo* holds, then the step from γ to γ' is possible") with the conditional operator `if .. then .. else ..`:
 $\gamma \rightarrow \text{if } \text{hypo} \text{ then } \gamma' \text{ else } \gamma''$

where γ'' has to be defined appropriately. But of course this is only possible when there are no extra variables in *hypo*, which is an exceptional case (e. g. both (1) and (2) contain extra variables, viz. σ_1 and stmt'_1). Furthermore, the problem of defining a suitable γ'' is not trivial (we will return to this problem).

So we have to be a little bit more inventive and have to find a way of disposing of the extra variables. Consider rule (1). The extra variable σ_1 stands for a terminal configuration that is related to $\langle \text{stmt}_1, \sigma \rangle$ by the transition relation. Viewing this relation more operationally, we can rephrase this as σ_1 standing for a possible (one-step) *result* of evaluating $\langle \text{stmt}_1, \sigma \rangle$.⁴ The name σ_1 itself is irrelevant; we only need the property that it denotes a terminal configuration.

$\langle \text{stmt}_1, \sigma \rangle$ does not contain extra variables; so it may safely occur on the right-hand side of the rewrite rule that we are aiming at. Since we are interested in its result, we enclose it by an additional operator *eval* that is intended to yield the result of evaluating its argument. By using *let* abstraction, i. e. λ -terms, we can name this result σ_1 , and we arrive at the rewrite rule

$$\langle \text{stmt}_1; \text{stmt}_2, \sigma \rangle \rightarrow \text{let } \sigma_1 = \text{eval}(\langle \text{stmt}_1, \sigma \rangle) \text{ in } \langle \text{stmt}_2, \sigma_1 \rangle \quad (3)$$

Note that σ_1 , although not appearing on the left-hand side, is not an extra variable. It is a bound variable of λ -calculus and, as much as term rewriting is concerned, it is just a constant of type T' . We assume that *let* terms are evaluated in applicative order (in *call by value* fashion).

So far, this looks like the kind of rule we wanted. But there still remains a problem. We have the other rule (2), and when we apply our procedure to this rule, we end up with a rewrite rule like

$$\langle \text{stmt}_1; \text{stmt}_2, \sigma \rangle \rightarrow \text{let } cf = \text{eval}(\langle \text{stmt}_1, \sigma \rangle) \text{ in } \langle cf \downarrow 1; \text{stmt}_2, cf \downarrow 2 \rangle \quad (4)$$

where cf is a variable of type $\text{Stmt} \times \Sigma$ and $\downarrow 1$ and $\downarrow 2$ are the projections to the first and second component of a tuple, respectively. Now the left-hand sides of (3) and (4) are identical, and each of the two rules can be applied in any case where the other could be applied, too. In (1) and (2), the decision which rule to apply is taken in the hypothesis by means of a type check. In order to get correct rewrite rules, we must add this kind of check as well: We must test whether the result of evaluating $\langle \text{stmt}_1, \sigma \rangle$ is terminal or not. And for the case that the result is non-terminal even though we chose the rule derived from (1),

⁴ There may be more than one possible result if the language is non-deterministic.

we must provide a “way back” giving a result that still allows application of the other rule: choosing the wrong rule must not lead into a “dead end”.

Implementing the type check is simple: it amounts to having rules of the form:

$$\langle stmt_1; stmt_2, \sigma \rangle \rightarrow \\ \text{let } \sigma_1 = eval(\langle stmt_1, \sigma \rangle) \text{ in if } type(\sigma_1) = T \text{ then } \langle stmt_2, \sigma_1 \rangle \text{ else } \dots \quad (5)$$

$$\langle stmt_1; stmt_2, \sigma \rangle \rightarrow \\ \text{let } cf = eval(\langle stmt_1, \sigma \rangle) \text{ in} \\ \text{if } type(cf) = Stmt \times T \text{ then } \langle cf \downarrow 1; stmt_2, cf \downarrow 2 \rangle \text{ else } \dots \quad (6)$$

More problematic is the “way back” that must be placed in the `else` parts of (5) and (6). Intuitively, we would demand that in these cases, the original configuration $\langle stmt_1; stmt_2, \sigma \rangle$ should remain unchanged. But we cannot simply put this into the `else` parts since it would render the rewrite system non-terminating: If the type check failed, the same rule could be applied over and over again.

So we must find a way to indicate that a rewrite rule has been tried in vain (i. e. its type check has been rewritten to false). For each of the rules generated from the SOS rules there must be a flag that can be raised when the `else` part is selected. This, however, is exactly the kind of situation that the concept of contexts has been defined for. Configuration terms never occur in nested form on the left-hand side of rewrite rules, so we can supply each of these with an appropriate context.

In our example, there are only two rules. Hence it suffices to introduce contexts as elements of $\{0, 1, *\} \times \{\text{on}, \text{off}\}^2$ and the desired rewrite rules become (for the one-step case)

$$\langle stmt_1; stmt_2, \sigma \rangle @ \langle 1, \text{on}, s \rangle \rightarrow \\ \text{let } \sigma_1 = eval(\langle stmt_1, \sigma \rangle @ \langle 1, \text{on}, \text{on} \rangle) \text{ in} \\ \text{if } type(\sigma_1) = T \\ \text{then } \langle stmt_2, \sigma_1 \rangle @ \langle 1, \text{on}, \text{on} \rangle \\ \text{else } \langle stmt_1; stmt_2, \sigma \rangle @ \langle 1, \text{off}, s \rangle \quad (7)$$

$$\langle stmt_1; stmt_2, \sigma \rangle @ \langle 1, s, \text{on} \rangle \rightarrow \\ \text{let } cf = eval(\langle stmt_1, \sigma \rangle @ \langle 1, \text{on}, \text{on} \rangle) \text{ in} \\ \text{if } type(cf) = Stmt \times \Sigma \\ \text{then } \langle cf \downarrow 1; stmt_2, cf \downarrow 2 \rangle @ \langle 1, \text{on}, \text{on} \rangle \\ \text{else } \langle stmt_1; stmt_2, \sigma \rangle @ \langle 1, s, \text{off} \rangle \quad (8)$$

The rules that define the operator `eval` guarantee that its argument is evaluated appropriately (see section 5.1); so $\langle stmt_1, \sigma \rangle$ is evaluated in one step only. Furthermore, we can easily prove that the `else` parts are smaller than the left-hand sides (under the well-founded ordering `off < on`); there is no termination problem when the type check fails. So we see that (7) and (8) are rules of the kind we have been looking for. In the following, we will call them **SOS-derived rules**.

What remains is to mention what happens if the hypothesis contains simple Boolean conditions. The preconditions b_i can safely be put into the type check since they do not contain extra variables. The conditions B_k restricting the intermediate and final configurations must also become part of the type check with the extra variables being replaced by suitable selection expressions in the style that has been used in rule (8). And finally, multiple transitions in the hypothesis are translated into iterated let expressions.

5 Properties of the transformed system

5.1 The basic rewrite system \mathcal{B}

The purpose of our transformation process is to provide a way to simulate program executions as specified by the operational semantics with the help of rewriting sequences. Since we want to employ “pure” rewriting and not rewriting modulo some equational theory, we also have to supply rewrite rules for modelling properties of the underlying data types. These rules form the basic rewrite system \mathcal{B} . We require \mathcal{B} to allow all rewritings that are not directly connected to application of SOS rules. Especially, all conditions should be decidable by rewriting. This amounts to demanding that \mathcal{B} be complete and correct in the logical sense (w. r. t. the standard interpretation of logical symbols), and also complete, i. e. confluent and terminating, in the sense of term rewriting.⁵ So we have the general assumption that \mathcal{B} provides (in the logical sense) a correct and complete decision procedure for all conditions that do not depend on the semantics definition. This means that each term expressing such a condition has exactly one normal form w. r. t. \mathcal{B} , viz. either true or false.

The operator *eval* plays a special rôle. It only occurs in terms of the form $\text{eval}(\gamma @ k)$, and its purpose is to ensure that its argument configuration γ is evaluated according to the context k . This is achieved by retaining the *eval* and the context as long as further evaluation is needed; only configurations in a 0-context or a terminal configuration in a *-context are completely evaluated. So we have the rules

$$\begin{aligned}\text{eval}(\gamma @ \langle 0, \dots \rangle) &\rightarrow \gamma \\ \text{eval}(t @ \langle *, \dots \rangle) &\rightarrow t\end{aligned}$$

where γ is a variable for configurations and t for terminal configurations.

5.2 Simulation

The transformation procedure has been devised in order to produce a rewrite system \mathcal{R} that models a semantics definition \mathcal{S} as closely as possible, the characteristic feature being the set of possible transition sequences. Therefore the most interesting questions to ask about \mathcal{R} are what rewriting sequences are possible and how they are related to the transition sequences of \mathcal{S} . We will see that the

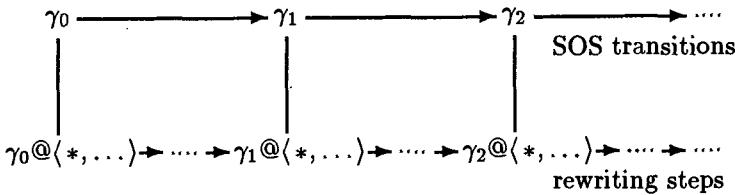
⁵ The practical consequences of a system not fulfilling these demands are discussed in [7].

relation between \mathcal{R} and \mathcal{S} is indeed very close; if rewriting is considered modulo the equational theory of the basic system \mathcal{B} , we have a 1-1 correspondence between rewriting sequences and “flattened” transition sequences (where steps needed to establish a premise are also visible).

Let \mathcal{S} be defined by the transition system $(\Gamma, T, \rightarrow_{\mathcal{S}})$, and let $\rightarrow_{\mathcal{S}}$ be given by a set of $N \in \mathbb{N}$ deduction rules. We will use the following additional abbreviations for contexts, where $k \in [N]$ and $r_j \in \{\text{on}, \text{off}\}$ for $j \in [N]$:

$$\begin{aligned} K_{0f} &=_{df} \langle 0, \text{on}, \dots, \text{on} \rangle \\ K_1^{(k)} &=_{df} \langle 1, r_1, \dots, r_{k-1}, \text{on}, r_{k+1}, \dots, r_N \rangle \\ K_f &=_{df} \langle *, \text{on}, \dots, \text{on} \rangle \\ K^{(k)} &=_{df} \langle *, r_1, \dots, r_{k-1}, \text{on}, r_{k+1}, \dots, r_N \rangle \end{aligned}$$

Overview: The simulation of \mathcal{S} by \mathcal{R} can be described as below. The intermediate terms in the rewriting sequence result from applying rules from \mathcal{R} to configuration terms. They need not themselves be configuration terms, but they are equal to such a term modulo $=_{\mathcal{R}}$.



Each transition step is modelled by a rewriting sequence in \mathcal{R} which generally has more than just one step. In order to be allowed to perform one step $\gamma_1 \rightarrow_{\mathcal{S}} \gamma_2$ using the transition system, we usually have to perform other transitions before that correspond to premises of transition rules. These “hidden” transitions only contribute indirectly to $\gamma_1 \rightarrow_{\mathcal{S}} \gamma_2$ by determining parts of γ_2 . So the transition process is not organized in linear form; each transition is equipped with a tree of other transitions (a proof tree) that justifies it. The corresponding rewriting process, however, can only construct flat sequences of terms. Therefore all the hidden transitions become part of the simulating sequence $\gamma_1 @ K_f \xrightarrow{*_{\mathcal{R}}} \gamma_2 @ K_f$ as well. Furthermore, rewriting makes use of the rules in \mathcal{B} , while transition takes place modulo $=_{\mathcal{B}}$.

Simulation works in the other direction as well. If we have a rewriting sequence that uses one SOS-derived rule, then this sequence corresponds to a transition sequence that is obtained via this particular SOS rule.

We have the following simulation results:

One-step completeness

$$\begin{aligned} \forall \gamma, \gamma' \in \Gamma \ \forall k \in [N] \ \forall r_1, \dots, r_{k-1}, r_{k+1}, \dots, r_N \in \{\text{on}, \text{off}\} : \\ \text{if } \gamma \rightarrow_{\mathcal{S}} \gamma' \text{ using rule } k \text{ then } \gamma @ K_1^{(k)} \xrightarrow{+_{\mathcal{R}}} \gamma' @ K_0f \end{aligned} \tag{9}$$

This is the basic building stone for the simulation results, viz. the simulation of one transition step by a rewriting sequence.

One-step correctness

$$\begin{aligned} \forall \gamma, \gamma' \in \Gamma \forall k \in [N] \forall r_1, \dots, r_{k-1}, r_{k+1}, \dots, r_N \in \{\text{on}, \text{off}\} : \\ \text{if } \gamma @ K_1^{(k)} \xrightarrow{\mathcal{R}} \gamma' @ K_0 f \text{ then } \gamma \xrightarrow{s} \gamma' \end{aligned} \quad (10)$$

The names “correctness” and “completeness” are used in the logical sense: Any transition corresponds to a rewriting sequence (completeness), and any rewriting sequence that contains one outermost application of an SOS-derived rule corresponds to a transition step (correctness). Note how the use of 1-contexts $K_1^{(k)}$ restricts rewriting to exactly one transition-related step.

Normal form completeness and correctness

$$\forall \gamma \in \Gamma \forall t \in T : \gamma \xrightarrow{*} s t \Leftrightarrow \gamma @ K_f \xrightarrow{*} \gamma @ K_f \quad (11)$$

Building up inductively from the one-step results, we can obtain simulation properties for longer transition sequences. One special case is of particular interest: sequences that end with a terminal configuration describe the complete evaluation of their initial configuration. Furthermore, expressions like $\gamma \xrightarrow{*} s t$ ($\gamma \in \Gamma, t \in T$) may occur in the premises of SOS rules.

The proofs for these results can be found in [6]. Because one-step and normal form transitions are intertwined via transitions in the premises of rules, all results must be proved by one simultaneous induction (on the number of applications of SOS-derived rules).

Divergence completeness

From one-step completeness, we immediately obtain that each infinite transition sequence corresponds to an infinite rewriting sequence. So non-termination is preserved by the rewrite system:

$$\begin{aligned} \forall \{\gamma^{(i)}\} \in \Gamma^{\mathbb{N}} : \\ (\forall i \in \mathbb{N} : \gamma^{(i)} \xrightarrow{s} \gamma^{(i+1)}) \Rightarrow (\forall i \in \mathbb{N} : \gamma^{(i)} @ K_f \xrightarrow{*} \gamma^{(i+1)} @ K_f) \end{aligned} \quad (12)$$

Divergence correctness

On the other hand, all infinite rewriting sequences correspond to “infinite behaviour” of the transition system. Because of the additional requirement in section 3 about terminating transition systems, the rewriting sequence keeps “coming back” to configurations, i. e. each tail of the sequence contains a configuration-context pair; therefore one-step correctness yields the existence of a corresponding infinite transition sequence:

$$\begin{aligned} \forall \gamma \in \Gamma \forall \{t^{(i)}\} \in T(\Sigma, V)^{\mathbb{N}} : t^{(1)} = \gamma @ K_f \wedge (\forall i \in \mathbb{N} : t^{(i)} \xrightarrow{\mathcal{R}} t^{(i+1)}) \\ \Rightarrow \exists \{\gamma^{(i)}\} \in \Gamma^{\mathbb{N}} \exists j : \mathbb{N} \rightarrow \mathbb{N} \text{ strictly monotonic} : \\ (\forall i \in \mathbb{N} : \gamma^{(i)} = (t^{(j(i))} \downarrow 1) \wedge \gamma^{(i)} \xrightarrow{s} \gamma^{(i+1)}) \end{aligned} \quad (13)$$

5.3 Confluence and termination

The system \mathcal{R} consists of two parts: the basic system \mathcal{B} and the system \mathcal{R}' containing the SOS-derived rules. As already mentioned in section 5.1, we assume

\mathcal{B} to be complete, i. e. confluent and terminating, so there are no problems with this part. But for \mathcal{R}' , the situation is totally different because these properties are completely determined by the semantics of the language L .

As we have seen in the previous section, every rewriting sequence in \mathcal{R} has a direct counterpart in \mathcal{S} and vice versa. This has immediate consequences for confluence and termination. Assume \mathcal{R} is terminating. This means that there is no configuration-context pair that is the initial point for an infinite rewriting sequence. Therefore there is also no configuration that starts an infinite transition sequence in \mathcal{S} . Obviously, this property is equivalent to L being a language that only contains terminating programs.

For confluence, the situation is very similar. Consider rewriting modulo the equational theory E generated by the basic system \mathcal{B} . Then the only rewrite rules that we need are those derived from the SOS system. Confluence of this rewrite system means that every configuration has at most one normal form (modulo E). As a consequence, for each initial state and each program starting in this state, there is at most one final state, and hence the programming language must be deterministic.⁶

So typically \mathcal{R} is not complete. In most cases, it will be non-terminating, and therefore normalization of configuration terms must be handled with care. Languages in the tradition of CSP ([16]) and Occam ([17]) do not even lead to confluent systems since they contain a non-deterministic choice operator. This might seem a serious drawback of the method, but it only reflects the desire to have a rewrite system that models the semantics as closely as possible. And the problem is very well known: Interpreters for functional languages, say, usually do not terminate when interpreting programs that are (semantically) “non-terminating”, disregarding restrictions like finite stack size.

There is also no point in completing the system \mathcal{R} , e. g. by applying the Knuth-Bendix procedure (cf. [19]). Completion would add new rules to the system, and for these rules there would be no counterpart in the original SOS system. So the simulation property would disappear; essentially, the result of completion corresponds to a language where all non-determinism has been artificially removed by declaring different results for one program as equal.

6 Application

The method presented has been successfully applied in solving a problem originating from the ProCoS project. For a language named PL_0^R , two different semantics definitions have been given (SOS and denotational), and the aim is to prove their equivalence. In [20], a standard mathematical hand proof is presented, its single steps mainly being based on induction on the structure of programs. A typical feature of such proofs is that subproofs are repeated in several places identically or only slightly modified, due to the similarity of semantics definitions for different language constructs. Therefore the use of an automated tool to check hand proofs or to assist in them is desirable.

⁶ This requirement can be slightly weakened; e. g. the evaluation order of parameters for function calls is unimportant as long as this evaluation has no side effects.

First results of applying the Larch Prover ([11]) to this problem are reported in [7]. By now, all the essential steps of the whole equivalence proof have been completed. For the basic idea behind application of the transformed rewrite system \mathcal{R} let us consider the subproblem of proving the equivalence of the semantics definitions of expressions (in a slightly simplified form).

Expressions $exp \in Expr$ are evaluated w. r. t. to an environment $\rho \in OpEnv$ and a state $\sigma \in \Sigma$, whose internal structure are not important here. The result is a value $v \in Val$. So configurations for the operational semantics are either tuples $\langle exp, \rho, \sigma \rangle \in Expr \times OpEnv \times \Sigma$ (non-terminal) or values from Val (terminal). The denotational semantics for expressions is given by a function $\mathcal{E} : Expr \times OpEnv \times State \rightarrow Val$, and we have to prove:

$$\forall exp \in Expr, \rho \in OpEnv, \sigma \in \Sigma : \langle exp, \rho, \sigma \rangle \xrightarrow{Expr} \mathcal{E}[exp] \rho \sigma \quad (14)$$

where \xrightarrow{Expr} is the SOS transition relation for expressions.

Now we transform \xrightarrow{Expr} into rewrite rules as described above, generating the system $\mathcal{R} = \mathcal{R}' \cup \mathcal{B}$. The denotational semantics is defined in form of equations that can directly be turned into rewrite rules; these rules belong to the basic system \mathcal{B} .

The proof that we have performed with the help of LP proceeds by induction on the structure of expressions. This form of induction is supported by LP; the necessary induction subgoals and hypotheses are generated automatically. For given exp, ρ and σ , the proof is structured as follows:

- First $\mathcal{E}[exp] \rho \sigma$ is evaluated using rules from \mathcal{B} ; this results in some term γ_1 .
- Next, the configuration/context term $t \equiv eval(\langle exp, \rho, \sigma \rangle @ \langle 1, on, \dots, on \rangle)$ is evaluated using all the rules in \mathcal{R} ; this results in some term γ_2 . In this phase, the induction hypotheses for the subexpressions of exp will also be used as rewrite rules.

Note that the rules for $eval$ (cf. section 5.1) together with the 1-context in t guarantee a one-step evaluation of $\langle exp, \rho, \sigma \rangle$.

- Finally, equality of γ_1 and γ_2 is proved using the rules of \mathcal{B} .

After the last step, we have proved

$$t \equiv eval(\langle exp, \rho, \sigma \rangle @ \langle 1, on, \dots, on \rangle) \xrightarrow{*_{\mathcal{R}}} \gamma_2 =_{\mathcal{B}} \gamma_1 \xleftarrow{*_{\mathcal{B}}} \mathcal{E}[exp] \rho \sigma$$

$eval$ operators are only introduced by the rules simulating the SOS transition rules. Thus γ_1 cannot contain such an operator, and there must be a point in the above rewriting sequence where it is removed from t . From the special form of the $eval$ elimination rules (cf. section 5.1), it follows that there must be an intermediate configuration term with a 0-context in the rewriting sequence (this term is the result of a successful application of a rule from \mathcal{R}'):

$$eval(\langle exp, \rho, \sigma \rangle @ \langle 1, on, \dots, on \rangle) \xrightarrow{+_{\mathcal{R}}} eval(\gamma'_2 @ \langle 0, \dots \rangle) \xrightarrow{*_{\mathcal{B}}} \gamma_2$$

The $eval$ elimination could not be applied in the first subsequence; so we have

$$\langle exp, \rho, \sigma \rangle @ \langle 1, on, \dots, on \rangle \xrightarrow{+_{\mathcal{R}}} \gamma'_2 @ \langle 0, \dots \rangle$$

and by one-step correctness (9) the rewriting proof turns out to be sufficient to prove the original goal (14) since the \rightarrow_B steps can be neglected (transitions with the SOS system take place modulo $=_B$).

Proofs about non-deterministic languages normally cannot be performed in a single step. For in such proofs, each possible result for a non-deterministic construct has to be considered, and this is most easily done by subsequently selecting all branches (by deleting those SOS rules that lead to other branches).

The Larch Prover turned out to be a suitable tool for implementing the transformed SOS rules since it is largely oriented towards easy formulation and application of rewrite rules. A major advantage of the system concerning our simulation is that normally intermediate results occurring in rewriting sequences are not displayed. This means that application of the rules in \mathcal{R}' remains completely hidden, and therefore the user is not confused by terms appearing during β -reduction or evaluation of type check conditions in SOS rules. Since the transformation of SOS rules into rewrite rules is very systematic, it was easy to implement a tool that generates LP input from SOS system descriptions. This tool has proved very helpful in the example proofs.

Another useful feature of LP's is that the user can control the rewriting process to a very large extent. E. g. the evaluation order can be changed (*inside-out* or *outside-in*), and rewrite rules can be prevented from being considered. This can be used to increase the efficiency of the system, since typically large groups of rules are known not to be applicable, and so the time for testing whether they match a given term can be saved.

Among the disadvantages of LP are the lack of powerful proof control mechanisms like strategies or tactics (as they are provided e. g. in HOL [13] or in KIV [15]) and its weak type concept. LP only supports a subset of many-sorted first-order logic. During our experiments, this never prevented any proofs, but it made their formulation a lot more complicated. In an order-sorted system like OBJ3 [12] or PVS [22], terms could be kept smaller⁷, and in a higher-order logic (present e. g. in HOL and PVS), many properties could be formulated much easier, since the rules about function application and extensionality would be supplied automatically. These rules can be simulated in LP's first-order logic, but a large number of explicit rules are needed for this.

7 Conclusion

In this paper, we have presented a way to simulate a special form of SOS definitions by standard term rewriting systems. Another approach to relating SOS definitions and equational logic is presented in [2]. Here an algorithm is described that transforms SOS rules into set of equations such that every true formula about the language can be proved in this theory. In [3] it is proved that for SOS definitions that only describe finite systems, these equations can be transformed into a complete term rewriting system. This technique, however, is restricted to SOS rules in a special format (GSOS) that is incomparable to the

⁷ If we have sorts $T_1 \subseteq T_2 \subseteq T_3$, we would like to be allowed to write $t \in T_3$ for all $t \in T_1$. In LP, however, we have to write the injections, e. g. *in-T*₃(*in-T*₂(*t*)).

format of definition 3. There exist other special rule formats, e. g. the *tyft/tyxt* format of [14], that have received special attention because of certain pleasant properties that such systems possess. These formats are less restrictive than the GSOS format, but such definitions have not yet been axiomatized by equations or rewrite rules. [8] shows how to model SOS definitions with other methods than term rewriting, using the facilities of the HOL system.

Other ways to combine λ -calculus and term rewriting are described in [18] and [10]. But both approaches extend the λ -calculus, and do not try to include some of its concepts in pure term rewriting.

Our method provides a very close simulation of transition rules by standard term rewriting rules. Since no additional formalism is needed, ordinary rewriting-based proof assistants can be used to implement it in reasoning about SOS definitions. In [7], we have described a successful example application where the Larch Prover [11] is used to assist in a proof of equivalence between an operational and a denotational semantics definition. During this experiment, it turned out that the rather complicated structure of the SOS-derived rules does not lead to difficulties. On the one hand, these rules can be generated automatically from the original SOS form, due to their systematic definition. And on the other hand, their application usually remains hidden in normalization processes. This means that the user of the proof tool need not worry about involved intermediate terms, but can concentrate on the logical structure of the proof.

Acknowledgements: I would like to thank the ProCoS group in Kiel, especially Bettina Buth, Yassine Lakhneche and Markus Müller-Olm, for their help in clarifying my ideas, Ursula Martin for drawing my attention to [1], and the anonymous referees for many helpful comments.

References

1. M. ABADI, L. CARDELLI, P.-L. CURIEN, AND J.-J. LÉVY. Explicit substitutions. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*, pages 31–46, 1990.
2. LUCA ACETO, BARD BLOOM, AND FRITS VAANDRAGER. Turning SOS rules into equations. In *Proceedings of the 7th IEEE Symposium on Logic in Computer Science, Santa Cruz, CA*, pages 113–124, 1992. Full version available as CWI Report CS-R 9218, Centrum voor Wiskunde en Informatica, Amsterdam, June 1992.
3. DOEKO BOSSCHER. Term rewriting properties of SOS axiomatisations. In *Proceedings of the Conference on Theoretical Aspects of Computer Science*, LNCS. Springer-Verlag, 1994. To appear.
4. JONATHAN BOWEN ET AL.. A ProCoS II project description: ESPRIT Basic Research project 7071. *Bulletin of the EATCS* 50, 128–137, 1993.
5. N. DE BRUIJN. Lambda-calculus notation with nameless dummies. *Indagationes Mathematicae* 34, 381–392, 1972.
6. KARL-HEINZ BUTH. Simulation of transition systems with term rewriting systems. Bericht 9212, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel, 1992.

7. KARL-HEINZ BUTH. Using SOS definitions in term rewriting proofs. In URSULA H. MARTIN AND JEANNETTE M. WING, editors, *Proceedings of the First International Workshop on Larch, Dedham, MA, 1992*, Workshops in Computing Series, pages 36–54. Springer-Verlag, 1993.
8. JUANITO CARMILLERI AND TOM MELHAM. Reasoning with inductively defined relations in the HOL theorem prover. Technical Report 265, University of Cambridge Computer Laboratory, August 1992.
9. NACHUM DERSHOWITZ AND JEAN-PIERRE JOUANNAUD. Rewrite systems. In JAN VAN LEEUWEN, editor, *Handbook of Theoretical Computer Science, Vol. B: Formal Models and Semantics*, chapter 6, pages 243–320. Elsevier/MIT Press, 1990.
10. DANIEL J. DOUGHERTY. Adding algebraic rewriting to the untyped lambda calculus. In RONALD V. BOOK, editor, *Proceedings of the 4th International Conference on Rewriting Techniques and Applications, Como, Italy*, LNCS 488, pages 37–48. Springer-Verlag, April 1991.
11. STEPHEN J. GARLAND AND JOHN V. GUTTAG. An overview of LP, the Larch Prover. In NACHUM DERSHOWITZ, editor, *Proceedings of the Third International Conference on Rewriting Techniques and Applications*, LNCS 355, pages 137–155. Springer-Verlag, 1989.
12. JOSEPH A. GOGUEN. OBJ as a theorem prover with applications to hardware verification. Technical Report SRI-CSL-88-4R2, SRI International, August 1988.
13. MICHAEL J. C. GORDON. HOL: A proof generating system for higher-order logic. In G. BIRTWISTLE AND P.A. SUBRAMANYAM, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer, 1988.
14. JAN FRISO GROOTE AND FRITS VAANDRAGER. Structured operational semantics and bisimulation as a congruence. *Information and Computation* 100(2), 202–260, October 1992.
15. MARITTA HEISEL, WOLFGANG REIF, AND WERNER STEPHAN. Tactical theorem proving in program verification. In MARK E. STICKEL, editor, *Proceedings of the 10th International Conference on Automated Deduction*, LNCS 449, pages 117–131. Springer-Verlag, 1990.
16. C. A. R. HOARE. *Communicating Sequential Processes*. Series in Computer Science. Prentice-Hall International, 1985.
17. INMOS LTD. *occam 2 Reference Manual*. Series in Computer Science. Prentice-Hall International, 1988.
18. STEFAN KAHR. λ -rewriting. PhD thesis, Fachbereich Mathematik und Informatik, Universität Bremen, January 1991.
19. DONALD E. KNUTH AND PETER B. BENDIX. Simple word problems in universal algebras. In J. LEECH, editor, *Proceedings of the Conference on Computational Problems in Abstract Algebra, Oxford, 1967*, pages 263–298. Pergamon Press, 1970.
20. YASSINE LAKHNECHE. Equivalence of denotational and structural operational semantics of PI_0^R . ProCoS Technical Report Kiel YL1, Christian-Albrechts-Universität Kiel, 1991.
21. GORDON D. PLOTKIN. An operational semantics for CSP. In DINES BJØRNER, editor, *Formal Description of Programming Concepts - II*, pages 199–225. North-Holland, 1983.
22. JOHN RUSHBY. A tutorial on specification and verification using PVS. In JAMES C. P. WOODCOCK AND PETER GORM LARSEN, editors, *Tutorial Material for FME '93: Industrial-Strength Formal Methods. Proceedings of the First International Symposium of Formal Methods Europe, Odense, Denmark*, pages 357–406, April 1993.

Strategies in Modular System Design by Interface Rewriting

S. Cicerone and F. Parisi Presicce*

Dipartimento di Matematica Pura ed Applicata,
Università de L'Aquila, I-67010 Coppito (AQ), Italy
e-mail: {cicerone, parisi}@vxscqaq.aquila.infn.it

Abstract. The problem of designing a modular system, using a set of predefined modules, with a given import and export interface has been reduced to the problem of generating a specification in an algebraic specification grammar. Here we tackle two important problems connected with the generation: the strategy to adopt in choosing the rewrite rules and the elimination of unnecessary searches. The first is investigated using a notion of similarity of specifications and a definition of value to guide the search algorithm; the second is solved using syntactical criteria (independent of the target specification) to determine that some derivation sequences are superfluous. The latter development has been influenced by similar work on graph grammars.

1 Introduction

The development of large correct software systems is very difficult without the appropriate support of notions such as modularization and interconnection of components [16,11,10]. In our context, a module specification [10,1,7] consists of four parts: a parameter part PAR to model genericity and parametrization (as in Ada generics, for example); an import part IMP (containing PAR) describing what the module needs from other modules (modelling a "virtual" module to be specified at a later time); an export interface EXP (containing PAR) specifying what part of the implemented functions are visible from the outside; and a body part BOD (containing all the others) with the description of how the functionalities exported (EXP) are implemented using those imported. Interconnection mechanisms for the horizontal structuring of systems are crucial for the stepwise development of large software in a flexible manner [7]. Interpreting the interconnections as operations on module specifications [1] it is easy to give a semantics to the main ones: *union* performed componentwise by specifying the common subcomponent to be identified; *composition* where the import of one module is matched with the export of another module; and *actualization* where the parameter part is replaced by an actual specification.

Module specifications designed and verified can be used via their interfaces, the only parts visible from the outside. A common problem is that of designing an interconnection of a predefined set of module specifications (of a library, for example) which realizes a given overall export interface from another given import interface. In [13,14] this problem has been addressed by viewing the visi-

* Current address: Dip. Scienze dell'Informazione , Univ. Roma "La Sapienza", via Salaria 113, I-00198 ROMA - Italy

ble part (PAR,IMP,EXP) of a module specification as a production of an algebraic specification grammar (ASG) [8], an extension of the algebraic theory of graph grammars [4] to structures other than graphs. In this approach, the applicability of a production ($IMP \leftarrow PAR \rightarrow EXP$) to a specification $SPEC$ to obtain a new specification $SPEC'$ indicates the existence of a module specification, obtained from the one which realizes the production, which has $SPEC$ and $SPEC'$ as import and export interface, respectively. A derivation sequence $PRE \Rightarrow SPEC_1 \dots \Rightarrow SPEC_n \Rightarrow GOAL$ can be automatically translated into the appropriate interconnection of the modules realizing the interfaces used as productions.

In general, given a specification $SPEC$ and a set LIB of productions, there may be several applicable productions, each with more than one occurrence of the left hand side in $SPEC$. The combinatorial explosion of possible sequences of derivations could be contained by analyzing beforehand the productions to remove from the search tree any path which will produce specifications already generated. This reduction is addressed in section 4, where syntactical criteria are given to predict the applicability of a rule after a derivation which uses another rule, and to avoid a derivation sequence which is equivalent to another derivation produced with a different order of the same productions. Many definitions and some results in this section are inspired by [12].

Having somewhat reduced the search tree, it is still necessary to have some criteria to choose (at least temporarily, trying to avoid backtracking) which production to use and which occurrence to apply. This is the topic of section 3, where we introduce the notion of *similarity* between two specifications and use it as a guide in selecting the appropriate occurrence. The search algorithm exploits the symmetry of the notion of direct derivation to develop a strategy based on the application of deductive (i.e., strict growth) rules in a forward fashion, and of deleting (i.e., strict reduction) rules in a backward fashion, interleaved with applications of the remaining productions.

The discussion in this paper is based on a standard notion of algebraic specification and a particular choice of specification morphism, needed to exploit some results in [5]. Some of the results can be extended to other situations, a few to a framework based on arbitrary institutions. All proofs, for lack of space, are omitted and can be found in [2].

2 Notation and Background

In this section we briefly review some basic notions of algebraic specifications ([6]) and of algebraic specification grammars ([13,14]).

Specifications

An algebraic specification (Σ, E) consists of a many sorted signature $\Sigma = (S, OP)$, and a set E of (positive conditional) equations. The three parts of a specification $SPEC = (S, OP, E)$ are referred to by using a subscript S_{SPEC} , OP_{SPEC} and E_{SPEC} . If $N \in OP$, $dom(N) \in S^*$ denotes the domain sorts and $cod(N) \in S$ the codomain sort of N . For $s \in S$, $N \in OP$, $e \in E$ we use:

- $SORT(N)$ as the union of $dom(N)$ and $cod(N)$;

- $OPNS(s)$ as the subset of OP containing the operations N with $s \in SORT(N)$;
- $OPNS(e)$ as the operations in the terms of e ;

- $EQNS(N)$ as the subset of E that contains equations e with $N \in OPNS(e)$

The notion of specification morphism $f : (\Sigma_1, E_2) \rightarrow (\Sigma_2, E_2)$ as a triple (f_S, f_{OP}, f_E) based on the accepted definition of signature morphism $f_\Sigma : \Sigma_1 \rightarrow \Sigma_2$, assures that the equations of E are labelled, and different labels eq_i may correspond to the same triple (X, t_1, t_2) representing the equation $eq_i : t_1 = t_2$; for $(eq_i : t_1 = t_2) \in E_1$ we have $(f_E(eq_i) : f^\#(t_1) = f^\#(t_2)) \in E_2$. We write *Specification* to denote the set of all specifications.

Specification Grammars

In the well known algebraic approach to Graph Grammars [4] it is possible to replace the category of graphs by the category of some other structure, giving rise to a new rewriting theory for high level structures [5].

Among those, a High Level Replacement (HLR) system was introduced in [13] in order to generate algebraic specifications using productions and derivations. An *algebraic specification production*, shortly SPEC-production, is an ordered pair $Pro = (IMP \leftarrow PAR \rightarrow EXP)$ of injective specification morphisms

$$\begin{array}{ccccc} IMP & \xleftarrow{i} & PAR & \xrightarrow{e} & EXP \\ \downarrow l & & \downarrow c & & \downarrow r \\ L & \xleftarrow{k} & CON & \xrightarrow{d} & R \end{array}$$

$i : PAR \rightarrow IMP$ and $e : PAR \rightarrow EXP$. A *direct derivation* consists of the two pushout diagrams of specifications. A production Pro is *applicable* to a specification L if there exist a morphism $l : IMP \rightarrow L$ and a *context* specification CON such that L is the pushout of IMP and CON . The result R of the derivation is pushout of EXP and CON . In this case we have the *direct derivation* $Pro : L \Rightarrow R$ (or $L \xrightarrow{Pro} R$), and we say that R is *derivable* from L via Pro . Notice that a direct derivation is symmetric and that if $Pro : L \Rightarrow R$, then $Pro^{-1} : R \Rightarrow L$ where $Pro^{-1} = (EXP \leftarrow PAR \rightarrow IMP)$. If $PROD$ is a set of SPEC-productions then the set of all the symmetric productions is denoted by $PROD^{-1}$. The specification morphisms that guarantee the existence of the pushout complement CON in a direct derivation are called *occurrence morphisms*. A morphism $l : IMP \rightarrow L$ is an occurrence if the following *Gluing Conditions* hold:

- a. $ID_S \cup DANG_S \subseteq i_S(S_{PAR})$
- b. $ID_{OP} \cup DANG_{OP} \subseteq i_{OP}(OP_{PAR})$, where
 - $ID_S = \{s \in S_{IMP} \mid \exists s' \in S_{IMP}, s \neq s', l_S(s) = l_S(s')\}$;
 - $ID_{OP} = \{N \in OP_{IMP} \mid \exists N' \in OP_{IMP}, N \neq N', l_{OP}(N) = l_{OP}(N')\}$;
 - $DANG_S = \{s \in S_{IMP} \mid \exists N \in OP_L \setminus l_{OP}(OP_{IMP}) \text{ and } l_S(s) \in SORTS(N)\}$;
 - $DANG_{OP} = \{N \in OP_{IMP} \mid \exists e \in E_L \setminus l_E(E_{IMP}) \text{ and } l_{OP}(N) \in OPNS(e)\}$.

The occurrence morphism $l : IMP \rightarrow L$ identifies in the direct derivation $Pro : L \Rightarrow R$ the L -part $DEL_l = l(IMP \setminus i(PAR))$, removed from L by the production Pro , and the R -part $INS_l = r(EXP \setminus e(PAR))$, glued by Pro to the context CON to realize the new specification R . Given a set $PROD$ of SPEC-productions we write $SPEC_1 \xrightarrow[PROD]^* SPEC_n$ to mean a sequence of $n \geq 0$ direct derivations $SPEC_1 \xrightarrow{P_1} SPEC_2 \xrightarrow{P_2} \dots \xrightarrow{P_n} SPEC_{n+1}$ with $P_i \in PROD$ for $i = 1, \dots, n$.

3 Strategy

The following notions are inspired by [12], where Graph Grammars are considered as models for rule-based systems in which solving state-space problems essentially requires searching in an exploding number of generated states which cannot be managed. The usual answer to this problem in AI is to prune the search-tree, selecting only some of the possible expansions of derivations. Any HLR system can be used to specify in a formal way many other similar problems. Algebraic specification grammars can model problems in which a transformation of a specification PRE by the rules in a library LIB to obtain a prefixed final specification $GOAL$, is required.

- Definition 1.** a). An (*AST*-)problem $P = (Ax, R, F)$ consists of a specification Ax , called axiom, a family of production rules R , and a unary predicate function on specifications F called filter. A *solution* for P is any Ax -reachable specification $SPEC$, i.e. $Ax \xrightarrow{P}^* SPEC$, for which $F(SPEC)$ is true.
- b). Given a class $PC = (\mathcal{AX}, R, F)$ of AST-problems, $P = (Ax, R, F)$ with $Ax \in \mathcal{AX}$, an algorithm S is called a *search algorithm* (w.r.t. PC), if and only if, when supplied with $Ax \in \mathcal{AX}$, the algorithm terminates with either a specification or the message FAIL. The result of S is correct if it is indeed a solution of P or FAIL otherwise.
- c). A *production-system* (PC, S) for AST-problems, briefly an AST-PS, consists of a class of AST-problems PC and a search algorithm S .

According to the basic execution model of production systems, derivations of a specification $SPEC$ based on a production rule Pro consists of two steps: retrieving the informations of how to apply Pro to $SPEC$, which can then be used to derive the new specification $SPEC'$ from $SPEC$. In terms of AST notions, this corresponds to two primitive functions:

Recognize : $Specification \times Rule \rightarrow OccurrenceSet$
 where $\text{Recognize}(SPEC, IMP \leftarrow PAR \rightarrow EXP) = \{g_1, g_2, \dots, g_m\}$ is
 the set of all occurrences $g_i : IMP \rightarrow SPEC$.

Derive : $Specification \times Occurrence \rightarrow Specification$
 where $\text{Derive}(SPEC, g) = SPEC'$ with $SPEC \not\sqsubseteq SPEC'$, assuming that
 an ‘occurrence’ carries the information about the corresponding rule.

Difficult problems arise in executing each of the two operations above, as well as in evaluating the filter F on specifications. The cost of the latter depends on the specific problem; the cost of the operations other than **Recognize** and **Derive** is ignored. The following notion of cost covers both, the problem of a search-space-reduction, and the efforts to determine the applicability of rules.

Definition 2. Given an AST-PS= (PC, S) , the cost of the search algorithm S is based on the primitive function **Recognize** and **Derive**; it is defined to be (N_1, N_2) with N_1 and N_2 being the number of calls for those functions. The cost (N_1, N_2) is said to be no greater than (N'_1, N'_2) iff $N_i \leq N'_i$ for $i = 1, 2$.

Since we do not want to adopt a specific search algorithm, we aim for a notion of optimization which guarantees that every search algorithm can be improved.

Definition 3. Given an AST-PS=(PC, S), and a search algorithm O w.r.t. PC , we call O an optimization w.r.t. S iff O yields the correct solution whenever S does and the cost of O is not greater than the cost of S .

The formalization can be given by the AST-problem $P = (PRE, LIB, F)$, where PRE is the predefined data type, LIB is the library of reusable modules defining the transformation rules, F the filter defined by:

$$\text{- } F(SPEC) = \text{true iff } SPEC = GOAL$$

An analysis of P could produce a search algorithm for PC (def. 1) independent of the initial axioms PRE . From the definition of P we observe that there is only one PRE -reachable specification that is a solution for P itself: a potential search algorithm for P should select in the search-tree a path from PRE to $GOAL$. If LIB has reasonable dimensions, it is impossible to think of selecting such a path visiting the tree in an exhaustive way; it is enough to observe that not only there exist several ways to transform each specification, but each rule in its own can generate many different results using all the occurrences selected by the `Recognize` primitive. For this reason it is difficult to work with search algorithms that exploit a backtracking going back for more than one level. Then, for each step the algorithm should check all the transformations, and then go down toward the specification that promises a 'better result'. It is necessary to find some criteria to assign a value showing the capability of each specification to lead to the final $GOAL$ specification. To this end we can assign a value to a specification according to the number of elements shared by $GOAL$.

3.1 Similarity

Definition 4. a) Let $SPEC_1 = (S_1, OP_1, E_1)$ and $SPEC_2 = (S_2, OP_2, E_2)$ be algebraic specifications and let (S, OP, E) be a subspecification of $SPEC_1$ such that $S \subseteq S_1$, $\emptyset \neq OP \subseteq OP_1$, $E \subseteq E_1$. A specification morphism $f : (S, OP, E) \rightarrow SPEC_2$ having injective components $f_S : S \rightarrow S_2$, $f_{OP} : OP \rightarrow OP_2$ and $f_E : E \rightarrow E_2$, is called *sharing morphism* of $SPEC_1$ into $SPEC_2$.

- b) In this case we call $SPEC_1$ *Partially Similar* to $SPEC_2$ and denote it by $SPEC_1 \xrightarrow{f} SPEC_2$.
- c) If the components of f are also surjective, we call $SPEC_1$ *Totally Similar* to $SPEC_2$ and denote it by $SPEC_1 \xrightarrow{f} SPEC_2$.
- d) The specification $SPEC_1$ is called *Comparison Specification* and $SPEC_2$ is called *Target Specification*. The subspecification (S, OP, E) and the elements in $SPEC_1 \setminus (S, OP, E)$ are denoted by *Core*($SPEC_1$) and *Remainder*($SPEC_1$) respectively, via the sharing morphism $f : (S, OP, E) \rightarrow SPEC_2$.

The set $M = \{f | SPEC_1 \xrightarrow{f} SPEC_2\}$ containing the sharing morphisms of $SPEC_1$ into $SPEC_2$ may have more than one element. To chose one we associate to the Target Specification $SPEC_2 = (S_2, OP_2, E_2)$, a mapping $c = (c_S, c_{OP}, c_E)$ that weighs sorts, operations and equations with $c_S : S_2 \rightarrow \mathbf{N}$, $c_{OP} : OP_2 \rightarrow \mathbf{N}$, $c_E : E_2 \rightarrow \mathbf{N}$

and define a function $val : M \rightarrow \mathbb{N}$ that evaluates the resources that the Comparison and the Target specifications share. For $f : (S, OP, E) \rightarrow SPEC_2 \in M$,

$$val(f) = \sum_{s \in S} c_S(f_S(s)) + \sum_{N \in OP} c_{OP}(f_{OP}(N)) + \sum_{e \in E} c_E(f_E(e))$$

is the *comparison value* of f . Among all the elements in M with the higher value, an arbitrary choice can selects the one we are looking for. If we choose f^* , it is called *main sharing morphism*, and the value $v(SPEC_1, SPEC_2, c) = val(f^*)$ denotes a measure of the similarity via some mapping c .

Definition 5. Let $SPEC_2$ be a Target specification with a weight mapping c .

a. the *Total Weight* of the Target is the value

$$p_T(SPEC_2) = \sum_{s \in S_2} c_S(s) + \sum_{N \in OP_2} c_{OP}(N) + \sum_{e \in E_2} c_E(e)$$

b. the function $Ratio_{SPEC_2, c} : Specification \rightarrow [0, 1]$ is given by

$$Ratio_{SPEC_2, c}(SPEC_1) = \frac{v(SPEC_1, SPEC_2, c)}{p_T(SPEC_2)}$$

where $SPEC_1$ denote a Comparison specification.

c. the specification $SPEC_*$ is the Optimal Comparison specification in $I \subset Specification$ w.r.t. the Target $SPEC_2$, when:

$$Ratio_{SPEC_2, c}(SPEC^*) = \max\{Ratio_{SPEC_2, c}(SPEC) | SPEC \in I\}$$

Fact 1. Let $SPEC_1$ be a Comparison specification and let $SPEC_2$ be a Target with a weight mapping $c = (c_S, c_{OP}, c_E)$. If $M = \{f | SPEC_1 \xrightarrow{f} SPEC_2\}$ then

$$Ratio_{SPEC_2, c}(SPEC_1) = 1 \iff \exists f \in M : SPEC_1 \xrightarrow{f} SPEC_2$$

The function *Ratio* could be the basis for an extension to a formal definition of a ‘metric’ that allows to measures the distance between specifications.

Example 1. . Let us suppose the following is a *Target Specification* in such a comparison between specifications with a mapping that uniformly weighs all the elements. It can be viewed as a specification of a system that represents a queue (FIFO), with some length, of weighted elements.

<u>QueueNat</u> =	
<u>sorts</u>	queue, nat, bool
<u>opns</u>	$NEW : \rightarrow queue$
	$QADD : queue \text{ nat} \rightarrow queue$
	$REMOVE : queue \rightarrow queue$
	$LENGTH : queue \rightarrow nat$
	$IS-EMPTY : queue \rightarrow bool$
	$ZERO : \rightarrow nat$
	$SUCC : nat \rightarrow nat$
	$TRUE : \rightarrow bool$
	$FALSE : \rightarrow bool$
	<u>eqns</u>
	For $q \in queue, n \in nat,$
	$e_1 : REMOVE(NEW) = NEW$
	$e_2 : REMOVE(QADD(q, n)) = IF$
	$IS-EMPTY(q) THEN NEW ELSE$
	$QADD(REMOVE(q), n)$
	$e_3 : IS-EMPTY(NEW) = TRUE$
	$e_4 : IS-EMPTY(QADD(q, n)) = FALSE$
	$e_5 : LENGTH(NEW) = ZERO$
	$e_6 : LENGTH(QADD(q, n)) =$
	$SUCC(LENGTH(q))$

The following are SPEC-productions defined by two module specifications. The first is $P_{MN} = (\text{MN-Imp} \xleftarrow{i_1} \text{MN-Par} \xrightarrow{e_1} \text{MN-Exp})$, where the three specifications are:

$$\begin{array}{lll} \underline{\text{MN-Imp}} = \underline{\text{MN-Par}} + & \underline{\text{MN-Par}} = & \underline{\text{MN-Exp}} = \underline{\text{MN-Imp}} + \\ \underline{\text{sorts}} \emptyset & \underline{\text{sorts}} \text{ nat} & \underline{\text{sorts}} \emptyset \\ \underline{\text{opns}} \text{ SUCC: nat} \rightarrow \text{nat} & \underline{\text{opns}} \text{ ZERO: } \rightarrow \text{nat} & \underline{\text{opns}} \text{ MUL: nat nat} \rightarrow \text{nat} \\ \underline{\text{eqns}} \emptyset & \underline{\text{eqns}} \emptyset & \underline{\text{eqns}} \emptyset \end{array}$$

while the second is $P_{IS} = (\text{IS-Imp} \xleftarrow{i_2} \text{IS-Par} \xrightarrow{e_2} \text{IS-Exp})$, with the specifications:

$$\begin{array}{lll} \underline{\text{IS-Imp}} = \underline{\text{IS-Par}} + & \underline{\text{IS-Par}} = & \underline{\text{IS-Exp}} = \underline{\text{IS-Imp}} + \\ \underline{\text{sorts}} \text{ string} & \underline{\text{sorts}} \text{ data} & \underline{\text{sorts}} \emptyset \\ \underline{\text{opns}} \text{ NIL: } \rightarrow \text{string} & \underline{\text{opns}} \emptyset & \underline{\text{opns}} \text{ INVERT: string} \rightarrow \\ \quad \text{LADD: data string} \rightarrow \text{string} & \underline{\text{eqns}} \emptyset & \quad \text{string} \\ \underline{\text{eqns}} \emptyset & & \underline{\text{eqns}} \emptyset \end{array}$$

Both the productions have inclusions as specification morphisms. Applying the productions P_{MN} and P_{IS} to the specification StackInt we obtain the direct derivations $P_{MN} : \text{StackInt} \Rightarrow \text{StackInt}'$ and $P_{IS} : \text{StackInt} \Rightarrow \text{InvertingStack}$ respectively.

$$\begin{array}{lll} \underline{\text{StackInt}} = & \underline{\text{InvertingStack}} = \underline{\text{StackInt}} + & \underline{\text{StackInt}}' = \underline{\text{StackInt}} + \\ \underline{\text{sorts}} \text{ stack, int} & \underline{\text{sorts}} \emptyset & \underline{\text{sorts}} \emptyset \\ \underline{\text{opns}} \text{ ZERO: } \rightarrow \text{int} & \underline{\text{opns}} \text{ INVERT: stack} \rightarrow & \underline{\text{opns}} \text{ MUL: int int} \rightarrow \\ \quad \text{SUCC: int} \rightarrow \text{int} & \quad \text{stack} & \quad \text{int} \\ \quad \text{PRED: int} \rightarrow \text{int} & \underline{\text{eqns}} \emptyset & \underline{\text{eqns}} \emptyset \\ \quad \text{EMPTY: } \rightarrow \text{stack} & & \\ \quad \text{PUSH: stack int} \rightarrow & & \\ \quad \quad \text{stack} & & \\ \underline{\text{eqns}} \emptyset & & \end{array}$$

Since $v(\text{StackInt}', \text{QueueNat}, 1) = 6/18$ and $v(\text{InvertingStack}, \text{QueueNat}, 1) = 7/18$, the production P_{IS} modifies StackInt making it more *similar* to the *target* than the result of the application $P_{MN} : \text{StackInt} \Rightarrow \text{StackInt}'$.

3.2 Search algorithm

In the context of similarity, the solution for P coincides with the Optimal Comparison specification in the set of all the specifications generated by the grammar $(\text{PRE}, \text{LIB}, \xrightarrow[\text{LIB}]{*})$ w.r.t. the target GOAL , when it is GOAL exactly. But the Optimal Comparison specification SPEC^* could be different from GOAL , either because of the search algorithm or because of a small library. In any case SPEC^* may be used in adapting the design of a partially designed modular system: instead of constructing a module with PRE and GOAL as interfaces, we need to implement only the elements that GOAL does not share with SPEC^* . We could also use the target weight mapping to increase the probability that a particular element could be in SPEC^* . We now give another AST-problem that is a formalization for our original problem.

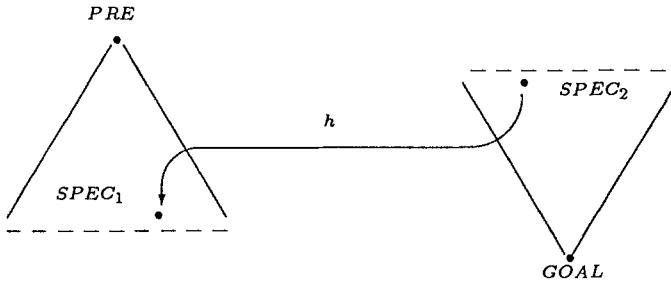
Definition 6. Define P' as the AST-problem $P' = (PRE, LIB, F')$, where PRE , LIB and GC are the same of P , whereas F' is defined as:

$$F'(SPEC_1) = \text{true} \text{ iff } \exists h : SPEC_2 \rightarrow SPEC_1 \wedge SPEC_2 \in (GOAL, LIB^{-1}, \xrightarrow[LIB^{-1}]{}^*)$$

The symmetric AST-problem of P' is $P'' = (GOAL, LIB^{-1}, F'')$, where

$$F''(SPEC_2) = \text{true} \text{ iff } \exists h : SPEC_2 \rightarrow SPEC_1 \wedge SPEC_1 \in (PRE, LIB, \xrightarrow[LIB]{}^*)$$

This new problem is important because it can be solved even if $GOAL$ cannot be derived from PRE . In fact, if $F'(SPEC_1) = \text{true}$ for some $SPEC_1 \in (PRE, LIB, \xrightarrow[LIB]{}^*)$, the situation can be represented as in the following picture:



with the search tree for the problem P' (the specification generated by the grammar $(PRE, LIB, \xrightarrow[LIB]{}^*)$), the tree that represents the specifications (including $SPEC_2$) generated by the grammar $(GOAL, LIB^{-1}, \xrightarrow[LIB^{-1}]{}^*)$ and the linking morphism h between the trees. By theorem 4.7 in [14], the derivation $PRE \xrightarrow[LIB]{}^* SPEC_1$ defines a module MOD_1 , whereas the derivation $GOAL \xrightarrow[LIB^{-1}]{}^* SPEC_2$, considered in a symmetric way, defines a module MOD_2 . The linking morphism h makes MOD_2 a client of MOD_1 , and allows their composition to obtain a module MOD that is a solution of the original problem.

The properties of P' are now studied to define a strategy on which a search algorithm for P' can be based. An initial measure of the difficulty to transform PRE into $GOAL$ can be given in a way independent of the knowledge contained in the library. The value $d(PRE, GOAL) = 1 - \text{Ratio}_{GOAL, 1}$ represents the percentage of the lacking resources of PRE with respect to the ones in $GOAL$. There exist some ordinary productions that allow ‘to deduct’ both the existence of further implicit resources in PRE and a surplus of elements in $GOAL$.

Definition 7. Let $Pro = (IMP \leftarrow PAR \rightarrow EXP)$ be a rule in LIB and let $SPEC$ be a specification.

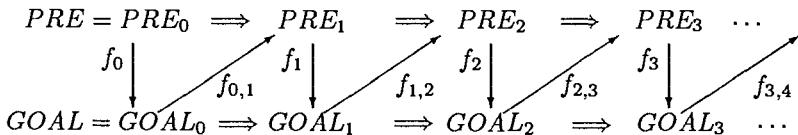
- i. $App(SPEC) = \{Pro \in LIB \mid \text{Recognize}(SPEC, Pro) \neq \emptyset\}$
- ii. $Ded(SPEC) = \{Pro \in App(SPEC) \mid PAR = IMP \neq EXP\}$

The deductive productions in the set $Ded(SPEC)$, when applied to $SPEC$ via an occurrence l , yield $DEL_l = \emptyset$ in forward mode and $INS_l = \emptyset$ in backward mode. So

it is possible to use a search algorithm starting with a *canonical phase* that tries to reduce the difficulty of transforming PRE into $GOAL$ by means of the deductive rules of LIB . In fact, by these rules, a forward derivation from PRE leads to an enriched specification PRE' , and a backward derivation from $GOAL$ to a final specification $GOAL'$ with a minimum number of elements to be implemented. An appropriate use of deductive rules assures $d(PRE', GOAL') \leq d(PRE, GOAL)$. Using the concepts of similarity we can now define the strategy for a first phase in a search algorithm for P' . During the enrichment of PRE (*growing phase*), if we apply the rules $Pro \in Ded(SPEC_i)$ to $SPEC_i$ via the occurrence $l \in \text{Recognize}(SPEC_i, Pro)$, giving rise to the direct derivation $SPEC_i \Rightarrow SPEC_{i+1}$, we can consider the difference

$$Info(Pro, l, SPEC_i) = v(SPEC_{i+1}, Target, c) - v(SPEC_i, Target, c)$$

This value represents the increase of the information that $SPEC_i$ shares with target, carried by the deductive Pro . As target, we can assume the final specification $GOAL$ or some other, as we will see later. For each step we choose the deductive one that carries the greatest increase; if no one can bring a positive increase, the growing phase stops. Analogously, during the *shrinking phase* from the final specification $GOAL$, at each step we can apply the deductive one that cuts the greatest number of elements in the remainder of $GOAL$ via some sharing morphism w.r.t. PRE (or w.r.t. the result PRE' of the growing phase). Since both growing and shrinking phases require calls to primitive *Recognize* and *Derive* in a proportional way to $d(PRE, GOAL)$, choosing $GOAL'$ as target for the comparisons in the forward derivation and also choosing PRE' as target for the ones in backward derivation, reduces the cost of the algorithm. Hence we prefer to adopt an *interaction* between forward and backward derivations.



In the above diagram the main sharing morphisms between the ‘current’ and target derivation for each derivation step are represented:

- $f_i : PRE_i \rightarrow GOAL_i$, identifying the resources that PRE_i shares with the target $GOAL_i$ before deriving PRE_{i+1} .
- $f_{i-1,i} : GOAL_{i-1} \rightarrow PRE_i$, identifying the resources that $GOAL_{i-1}$ shares with the target PRE_i before deriving $GOAL_i$.

The choice of starting the interactive derivation from PRE_0 is due to an immediate decrease of calls to primitive functions, rather than from $GOAL_0$, that produces benefits only on the length of forward derivation $PRE \Rightarrow PRE'$. However, for each step, the existence of a linking morphism can be tested, requiring the valuation of the filter of problem P' .

The following fact takes into consideration the cost of the filter valuation, allowing to focus on the current specifications.

Proposition 8. If PRE_i is the current derived specification in the canonical phase of a forward derivation, then:

$$\exists h : GOAL_{i-1} \rightarrow PRE_i \Rightarrow \exists h' : GOAL_k \rightarrow PRE_j \quad \forall k \leq i-1, \forall j \leq i$$

If $GOAL_i$ is the current derived specification in the canonical phase of a backward derivation, then:

$$\exists h : GOAL_i \rightarrow PRE_i \Rightarrow \exists h' : GOAL_k \rightarrow PRE_j \quad \forall k \leq i, \forall j \leq i$$

Any search algorithm for the problem P' (see def. 6) should have a Canonical Phase to construct its first part. At this point it is also necessary to take into consideration all the productions in LIB to define the strategy for deriving $GOAL'$ from PRE' . This post-canonicalization phase can be based on an interactive derivation again, but we need to modify the way to choose the rule to apply. A procedure can select among all the rules in $App(PRE')$, but not in $Ded(PRE')$, the production that causes the largest increase of shared resources with respect to the current target, in spite of the removed part $DEL_l \neq \emptyset$. If DEL_l contains some ‘useful’ elements, a new series of applications of deductive rules takes place, while another procedure can select among all the rules in $App(GOAL')$, but not in $Ded(GOAL')$, the symmetric production that causes the largest decrease of the elements not shared (those in the Remainder) with respect to the current target, in spite of the removed part $INS_l \neq \emptyset$. If INS_l contains some ‘useless’ elements, a new series of applications of deductive rules takes place.

4 Optimization

In this section we consider again the results of [12] in the theory of Graph Grammars, and present criteria that allow to optimize any search algorithm in the context of algebraic specification transformations. Our approach to optimization is based on properties of rules which must safely avoid calls to the corresponding primitive functions, thus reducing the cost of the algorithm.

The derivation bag $P_J(\mathcal{S})$ of a specification bag \mathcal{S} with respect to a family of rules $(P_j)_{j \in J}$ is $P_J(\mathcal{S}) = \{SPEC' \mid \exists SPEC \in \mathcal{S}, j \in J \text{ such that } SPEC \xrightarrow{P_j} SPEC'\}$. We also use the notation $P_j(SPEC)$ for $J = \{j\}$ and $\mathcal{S} = \{SPEC\}$.

Definition 9. A rule P_1 is k -monotonic with respect to a rule P_2 if and only if

$$\forall SPEC : |P_2(SPEC)| = k \xrightarrow{\text{implies}} (\forall SPEC_1 \in P_1(SPEC) \mid P_2(SPEC_1) | \leq k)$$

A rule P_1 is called monotonic w.r.t. a rule P_2 if and only if

$$\forall SPEC : P_2(SPEC) = \emptyset \xrightarrow{\text{implies}} P_2P_1(SPEC) = \emptyset$$

This definition gives rise to a simple improvement whenever we find those rules where the non-applicability of the second to the result of the first can be predicted, provided the second has been non-applicable before. Thus monotonicity allows to eventually skip some `Recognize`-calls.

Fact 2. Given rules $\{P_1, P_2, \dots, P_n\}$, each k -monotonic w.r.t. a rule P_q , for every $SPEC_{s_i} \in P_{s_i} \cdots P_{s_1}(SPEC)$, with $i \geq 1$, $s_j \in \{1, 2, \dots, n\}$, $j = 1, \dots, i$:

$$|P_q(SPEC)| = k \stackrel{\text{implies}}{\implies} |P_q(SPEC_{s_i})| \leq k$$

In order to let a search-algorithm take advantage of a *precomputation pass* which distinguishes rules which are monotonic, *effectively computable* criteria must be found. A **syntactical monotonicity criterion** is an effectively computable binary predicate on rules telling whether these rules are monotonic.

Given two syntactical monotonicity criteria SC and SC_b , the latter is said to be better if and only if $SC \subset SC_b$. A syntactical monotonicity criterion is said to be **optimal** if and only if there is no better syntactical monotonicity criterion.

Asking how an interaction of rules can effectively be characterized, we start by looking at the ways in which two rules may overlap in a derivation. To be more precise, we ask how the part in specification $SPEC_2$, defined by the intersection of the images of the occurrence morphisms $r_1 : EXP_1 \rightarrow SPEC_2$, $r_2 : IMP_2 \rightarrow SPEC_2$, can be characterized when $SPEC_1 \xrightarrow{p_1} SPEC_2 \xrightarrow{p_2} SPEC_3$.

Definition 10. (Gluing Relation Set)

Given two specification morphisms $e_1 : PAR_1 \rightarrow EXP_1$, $e_2 : PAR_2 \rightarrow IMP_2$, the **gluing relation set** is the set $\tilde{GRS}(e_1, e_2)$ of relations

$$\tilde{gr} = (\tilde{gr}_S \subseteq S_{EXP_1} \times S_{IMP_2}, \tilde{gr}_{OP} \subseteq OP_{EXP_1} \times OP_{IMP_2}, \tilde{gr}_E \subseteq E_{EXP_1} \times E_{IMP_2})$$

on sorts, operations and equations, such that each $\tilde{gr} \in GRS(e_1, e_2)$ also satisfies the following axioms.

For $a, a' \in EXP_1$, $b, b' \in IMP_2$, $\bar{a} \in EXP_1 \setminus e_1(PAR_1)$, $\bar{b} \in IMP_2 \setminus e_2(PAR_2)$:

(Ax1). $a \tilde{gr}_{OP} b \Rightarrow (cod(a) \tilde{gr}_S cod(b))$ and "every $x \in dom(a)$ bijectively corresponds to a $y \in dom(b)$ such that $x \tilde{gr}_S y$ "

(Ax2). $a \tilde{gr}_E b \Rightarrow MATCH(a, b) = TRUE$

(Ax3). $a \tilde{gr}_S \bar{b} \Rightarrow \forall N \in OPNS(a) \exists N' \in OPNS(\bar{b}) \text{ and } N \tilde{gr}_{OP} N'$

(Ax4). $\bar{a} \tilde{gr}_S b \Rightarrow \forall N' \in OPNS(b) \exists N \in OPNS(\bar{a}) \text{ and } N \tilde{gr}_{OP} N'$

(Ax5). $(a \tilde{gr}_S \bar{b}) \text{ and } (a \tilde{gr}_S b') \Rightarrow \bar{b} = b'$

(Ax6). $(\bar{a} \tilde{gr}_S b) \text{ and } (a' \tilde{gr}_S b) \Rightarrow \bar{a} = a'$

(Ax7). $(a \tilde{gr}_{OP} \bar{b}) \text{ and } (a \tilde{gr}_{OP} b') \Rightarrow \bar{b} = b'$

(Ax8). $(\bar{a} \tilde{gr}_{OP} b) \text{ and } (a' \tilde{gr}_{OP} b) \Rightarrow \bar{a} = a'$

(Ax9). $a \tilde{gr}_{OP} \bar{b} \Rightarrow \forall e_1 \in EQNS(a) \exists e_2 \in EQNS(\bar{b}) \text{ and } e_1 \tilde{gr}_{OP} e_2$

(Ax10). $\bar{a} \tilde{gr}_{OP} b \Rightarrow \forall e_2 \in EQNS(b) \exists e_1 \in EQNS(\bar{a}) \text{ and } e_1 \tilde{gr}_{OP} e_2$

The function *MATCH* verifies that the two equations can be translated into each other via the identifications of the relation \tilde{gr}_{OP} .

A relation $\tilde{gr} \in \tilde{GRS}$, with $\tilde{gr} \subseteq e_1(PAR_1) \times e_2(PAR_2)$ is called an *Interface* relation.

Proposition 11. Given two specification morphisms $e : PAR_1 \rightarrow EXP$, $i : PAR_2 \rightarrow IMP$, each element $\tilde{gr} \in \tilde{GRS}(e, i)$ identifies an algebraic specification $SPEC_{\tilde{gr}} \in SPEC_{\tilde{GRS}}$.

There are two ‘projection’ morphisms $\pi_1 : SPEC_{\tilde{gr}} \rightarrow EXP$ and $\pi_2 : SPEC_{\tilde{gr}} \rightarrow IMP$ defined by:

$$- \forall (x, y) \in SPEC_{\tilde{gr}} : \pi_1((x, y)) = x, \pi_2((x, y)) = y.$$

If we replace the elements $SPEC_{\tilde{gr}}$ in $SPEC_{GRS}$ by the element $(SPEC_{\tilde{gr}}, \pi_1, \pi_2)$, we obtain a new set, denoted by PB_{GRS} .

Proposition 12. *Given two rules*

$$\begin{aligned} P_1 &= (IMP_1 \xleftarrow{i_1} PAR_1 \xrightarrow{e_1} EXP_1) \text{ and } P_2 = (IMP_2 \xleftarrow{i_2} PAR_2 \xrightarrow{e_2} EXP_2) \\ \text{the set } PB_{GRS} &\text{ contains exactly the pullbacks of all } EXP_1 \xrightarrow{r_1} SPEC_1 \xleftarrow{l_1} IMP_1, \\ &\text{with } r_1 \text{ and } l_1 \text{ occurrence morphisms.} \end{aligned}$$

The following Match Theorem is a special case of the Concurrency Theorem for HLR-Systems, which holds for the particular choice of specification morphisms reviewed in section 2 ([9]).

Theorem 13. *Given the sequence of derivations*

$$(S1) \quad SPEC_1 \xrightarrow{P_1} SPEC_2 \xrightarrow{P_2} SPEC_3$$

*there exists a $SPEC$ -derivation with the match-production $P_1 *_M P_2$*

$$(S2) \quad P_1 *_M P_2 : SPEC_1 \Rightarrow SPEC_3$$

called matched derivation of the sequence (S1).

*Viceversa each direct $SPEC$ -derivation (S2), using $P_1 *_M P_2$ leads to a derivation sequence (S1) using P_1 and P_2 .*

Now we can use the set of all gluing relations to characterize the way in which rules may overlap.

Lemma 14. *Given $P_1 = (IMP_1 \xleftarrow{i_1} PAR_1 \xrightarrow{e_1} EXP_1)$ and $P_2 = (IMP_2 \xleftarrow{i_2} PAR_2 \xrightarrow{e_2} EXP_2)$ and subspecifications $S_E \subseteq EXP_1$, $S_I \subseteq IMP_2$, then the proposition*

$$\forall SPEC_1 \xrightarrow{P_1} SPEC_2 \xrightarrow{P_2} SPEC_3, \quad r_1(S_E) \cap l_2(S_I) = \emptyset$$

with $r_1 : EXP_1 \rightarrow SPEC_2$ and $l_2 : IMP_2 \rightarrow SPEC_2$, is equivalent to

$$\forall \tilde{gr} \in GRS(e_1, i_2), \quad \tilde{gr} \not\subseteq S_E \times S_I$$

Using this lemma, potential overlapping of occurrences, defined as universally quantified propositions over an infinite number of specifications, are effectively decidable. In fact, since each of the specifications $EXP_1 = (S_1, OP_1, E_1)$ and $IMP_2 = (S_2, OP_2, E_2)$ is finite and so are the sets $S = S_1 \times S_2$, $OP = OP_1 \times OP_2$, $E = E_1 \times E_2$ and the powersets $\mathcal{P}(S)$, $\mathcal{P}(OP)$ and $\mathcal{P}(E)$, all \tilde{gr} -axioms can be checked in a finite number of steps. The classical notion of *parallel iterdependency* leads to a syntactical monotonicity criterion.

Definition 15. A rule P_1 is said to be **S-independent** of a rule P_2 if and only if

$$\forall EXP_1 \xrightarrow{r_1} SPEC \xleftarrow{l_2} IMP_2 : r_1(EXP_1) \cap l_2(IMP_2) \subseteq r_1(e_1(PAR_1)) \cap l_2(i_2(PAR_2))$$

where r_1 and l_2 satisfy the gluing conditions.

Fact 3. *S-indipendence is a syntactical k-monotonicity criterion.*

Unfortunately, s-indipendence is only a weak monotonicity criterion, since it is only a sufficient criterion: there could be a pair of rules P_1 and P_2 such that P_1 is monotonic w.r.t. P_2 , although P_1 is not s-indipendent of P_2 .

Lemma 16. *Given rules P_1 , P_2 and P_3 ,*

$$\left. \begin{array}{l} \forall \tilde{gr} \in \tilde{GRS} \text{ and } P_* = P_1 *_M P_2 = IMP_* \xleftarrow{i_*} PAR_* \xrightarrow{e_*} EXP_* \\ \text{with } M = (SPEC_{\tilde{gr}}, \pi_1, \pi_2) \in PB_{\tilde{GRS}} \\ \text{there is an occurrence morphism } g_3^+ : IMP_3 \rightarrow IMP_* \\ \text{such that } (g_{3S}^+(NGl_{3S}) \subseteq NGl_{*S} \wedge g_{3OP}^+(NGl_{3OP}) \subseteq NGl_{*OP}) \} (UCC) \end{array} \right\} (SintR)$$

is equivalent to

$$\forall SPEC (\exists SPEC \xrightarrow{P_1} SPEC_1 \xrightarrow{P_2} SPEC_2 \xrightarrow{\text{implies}} \exists SPEC \xrightarrow{P_3} SPEC_3) \} (SemR)$$

Remark. – In a production $P = (IMP \xleftarrow{i} PAR \xrightarrow{e} EXP)$, the elements of the set $Gl = i(PAR)$ are called *Gluing Elements*, while $NGl = IMP \setminus i(PAR)$ contains the *Non-Gluing Elements*.

- (SyntR) is a shorthand for *Syntactical Relation*, whereas (UCC) stands for *Uncriticalness Condition*. The *Semantical Relation* (SemR) can be read as: ‘if P_1 and P_2 can sequentially be applied to $SPEC$, then P_3 must be applicable to $SPEC$ ’ and is equivalent to $P_3(SPEC) = \emptyset \xrightarrow{\text{implies}} P_2P_1(SPEC) = \emptyset$

Definition 17. Given two rules P_1 and P_2 , P_2 is **M-independent** of P_2 , if and only if for each non-Interface relation $\tilde{gr} \in \tilde{GRS}$ there is an occurrence morphism $g_2^* : IMP_2 \rightarrow IMP_*$ with $P_* = P_2 *_M P_1 = IMP_* \xrightarrow{i_*} PAR_* \xrightarrow{e_*} EXP_*$ and $M = (SPEC_{\tilde{gr}}, \pi_1, \pi_2) \in PB_{\tilde{GRS}}$, such that the *Uncriticalness Condition* ($g_{2S}^*(NGl_{2S}) \subseteq NGl_{*S} \wedge g_{2OP}^*(NGl_{2OP}) \subseteq NGl_{*OP}$) holds.

Theorem 18. *M-independence is an optimal syntactical monotonicity criterion.*

Any addition of correct software in any library should modify the information about the M-indipendence among all induced rules. In this way, each search algorithm could exploit search-space reduction, as well as a reduction of efforts to determine the applicability of rules. Along the lines of [12], there is a notion of *semi-commutativity* of P_1 w.r.t. P_2 which allows to interchange their applications. It can be shown that *S-indipendence* is a syntactical semi-commutativity criterion and that there is an optimal semi-commutativity criterion, called SC-independence. For lack of space, we refer to [2] for formal definition and proofs.

Example 2. . Given the specification morphism e_1 and i_2 , we illustrate the elements of the set $\tilde{GRS}(e_1, i_2) = \{\tilde{gr}_0, \tilde{gr}_1\}$, with:

- $\tilde{gr}_0 = (\emptyset, \emptyset, \emptyset)$
- $\tilde{gr}_1 = (\{(nat, data)\}, \emptyset, \emptyset)$.

The \tilde{gr} relation $\tilde{gr}_2 = (\{(nat, string)\}, \emptyset, \emptyset)$ is not contained in $\tilde{GRS}(e_1, i_2)$ because it does not verify (Ax.3). Some \tilde{gr} with $\tilde{gr}_{OP} \neq \emptyset$ should include the pair $(ZERO, NIL)$, but this requires the validity of $nat \tilde{gr}_S string$. Both \tilde{gr}_0 and \tilde{gr}_1 are Interface relations.

The set $\tilde{GRS}(e_2, i_1)$ is $\{\tilde{gr}'_0, \tilde{gr}'_1, \tilde{gr}'_2\}$, in which

- $\tilde{gr}'_0 = (\emptyset, \emptyset, \emptyset)$
- $\tilde{gr}'_1 = (\{(data, nat)\}, \emptyset, \emptyset)$
- $\tilde{gr}'_2 = (\{(string, nat)\}, \{(NIL, ZERO), (INVERT, SUCC)\}, \emptyset)$

and \tilde{gr}'_0 and \tilde{gr}'_1 are Interface relations.

The production P_{MN} is M-independent of the production P_{IS} , because of all the relations in $\tilde{GRS}(e_1, i_2)$ are Interface relations. Then, by the previous theorem:

$$\forall SPEC : P_{IS}(SPEC) = \emptyset \stackrel{\text{implies}}{\Rightarrow} P_{IS}P_{MN}(SPEC) = \emptyset$$

Viceversa, P_{IS} is not M-independent of P_{MN} ; in fact for the only not Interface relation \tilde{gr}_2 not exist such an occurrence morphism from $MN\text{-Imp}$ to IMP_* when $P_{IS} *_M P_{MN} = (IMP_* \leftarrow PAR_* \rightarrow EXP_*)$ and $M = (SPEC_{\tilde{gr}'_2}, \pi_1, \pi_2) \in PB_{\tilde{GRS}(e_1, i_2)}$.

5 Concluding Remarks

In this paper we have addressed the problem of deriving a given specification in an algebraic specifications grammar. Elsewhere [13,14], it has been shown that if the productions of the specification grammar are the interfaces of module specifications, then a derivation sequence from the initial specification of the grammar to the objective specification can be automatically translated into an interconnection of the corresponding module specifications. Even for small libraries of modules, the search space for the problem of deriving a specification can be intractably large. We have found syntactical criteria to prune the search tree by analyzing only the interaction of the different productions, independently of the specification to be generated. So, if two productions P_1 and P_2 are, say, commutative, then only one of the two sequences P_1P_2 and P_2P_1 is considered. To guide the search in the pruned tree, we have used the notion of similarity, to measure the distance between two specifications. In choosing the appropriate occurrence of a production, the total weight of a morphism is used, defined in terms of an arbitrary importance map defined on the goal specification.

One of the objectives of this work is to produce an "automatic helper" to assist in the design of a modular system from a library (typically a prototype to investigate the feasibility and to validate the adequacy of the goal specification). What has been developed does not depend on the notion of module specification chosen, but can be used in any context where productions of algebraic specifications are used [15]. While most of section 4 depends on syntactical criteria based on the intrinsic

structure of the algebraic specifications, the development in section 3 is based on the notion of similarity and of weight of an occurrence, both defined essentially in terms of morphisms and therefore directly extendable to institutions other than the one used (essentially to simplify the presentation).

References

1. E. K. Blum, H. Ehrig, F. Parisi-Presicce, *Algebraic Specification of Module and their Interconnections*, J. Comp. System Sci. 34, 2/3, 1987, 239-339.
2. S. Cicerone, F. Parisi-Presicce: *Strategies in Modular System Design by Interface Rewriting*, Technical Report N. 39/93, Dip. Matematica Pura ed Applicata, Univ. L'Aquila, 1993.
3. S. Cicerone, F. Parisi-Presicce: *On the Complexity of Specification Morphism*, Technical Report N.32/93, Dip. Matematica Pura ed Applicata, Univ. L'Aquila, 1993.
4. H. Ehrig: *Introduction to the Algebraic Theory of Graph Grammars*, LNCS 73, 1-69, 1979.
5. H. Ehrig, A. Habel, H.-J. Kreowski, F. Parisi-Presicce: *From Graph Grammars to High-Level Replacement System*, Proc. 4 Int. Workshop on Graph Grammars and Application to Comp. Sci., LNCS 532, 1991, 269-291.
6. H. Ehrig, B. Mahr: *Fundamentals of Algebraic Specification 1: Equation and Initial Semantics*, EATCS Monographs on Theoret. Comp. Sci., vol. 6, Springer-Verlag, 1985.
7. H. Ehrig, B. Mahr: *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints*, EATCS Monographs on Theoret. Comp. Sci., vol. 21, Springer-Verlag, 1990.
8. H. Ehrig, F. Parisi-Presicce: *Algebraic Specification Grammars: A Junction Between Module Specification and Graph Grammars*, Proc. 4 Int. Workshop on Graph Grammars and Application to Comp. Sci., LNCS 532, 1991, 292-310.
9. H. Ehrig, F. Parisi-Presicce: *High-Level Replacement System for Equational Algebraic Specification*, Proc. 3rd Int. Conf. Algebraic and Logic Programming, LNCS 632, 1992, 3-20.
10. H. Ehrig, H. Weber: *Algebraic Specification of Modules*, in 'Formal Models in Programming' (E.J. Neuhold, G. Chomist, eds.), North-Holland, 1985.
11. J. A. Goguen, J. Meseguer: *Universal Realization, Persistent Interconnection and Implementation of Abstract Modules*, LNCS 140, 1982, 265-281.
12. M. Korff: *Application of Graph Grammars to Rule-Based System*, Proc. 4 Int. Workshop on Graph Grammars and Application to Comp. Sci., LNCS 532, 1991, 505-519.
13. F. Parisi-Presicce: *A Rule-Based Approach to Modular System Design*, Proc. 12 Int. Conf. Soft. Eng., Nice(France), 1990, 202-211.
14. F. Parisi-Presicce: *Foundation of Rule-Based Design of Modular System*, Theoretical Comp. Science 83, 1991, 131-155.
15. F. Parisi-Presicce: *Reusability of Specifications and Implementations*, Proc. 2nd Int. Conf. on Alg. Method and Soft. Techn., AMAST '91, (M. Nivat, T. Rus, G. Scallo, C. Rattray eds.), Springer-Verlag 1992, 43-56.
16. D. L. Parnas: *A Technique for Module Specification with examples*, Comm. ACM 15, 5, 1972, 330-336.

Symbolic Model Checking and Constraint Logic Programming: a Cross–Fertilization

M.-M. Corsini, A. Rauzy

LaBRI, URA CNRS 1304 – Université Bordeaux I
351, cours de la Libération,
33405 Talence Cedex FRANCE
e-mail: {corsini, rauzy}@labri.u-bordeaux.fr

Abstract. In this paper, we present the constraint language **Toupie** which is a finite domain μ -calculus interpreter that uses extended decision diagrams to represent relations and formulae. “Classical” constraint logic programming languages over finite domains ($CLP(\mathcal{FD})$) are designed to find one solution to a constraint problem, eventually the best one according to a given criterion. In **Toupie**, constraints are used to characterize existing relationships between variables. We advocate the use of this paradigm to model and solve efficiently difficult constraint problems that are not tractable with $CLP(\mathcal{FD})$ languages.

Keywords: Symbolic Model Checking, Constraint Languages

1 Introduction

Constraint Logic Programming (CLP) has shown to be a very attractive field of research over recent years, and languages such as $CLP(\mathcal{R})$ [JL87], CHIP [Hen90] and PrologIII [Col90] have proved that this approach opens Logic Programming to a wide range of real life problems.

Languages of the family $CLP(\mathcal{FD})$, with constraints over finite domains, are based on the paradigm enumeration/propagation. They are mainly designed to find one solution to a given problem, eventually the best one according to some criterion (objective function). They use widely algebraic properties of the underlying domain, i.e. the set of relative numbers.

In this paper, we present the constraint language **Toupie** which is based on a different paradigm: constraints are mainly symbolic and are used to characterize relationships existing between variables. Namely, **Toupie** implements an extention of the propositional μ -calculus to finite domains. The propositional μ -calculus is a language designed to model the behavior of systems of concurrent processes, where μ denotes a least fixpoint operator used to describe properties of finite state machines.

In addition to the classical functionalities of symbolic finite domain constraint languages, a full universal quantification is available in **Toupie** and one can define relations (predicates) as fixpoints of equations.

This gain in expressiveness is coupled with a practical efficiency that comes from the management of the relations via decision diagrams:

- Decision diagrams encode relations in a very compact manner (thanks to the sharing of the subtrees).
- The algorithm that computes logical operations between two decision diagrams uses a learning mechanism: the more computations it has performed, the more it is efficient.

The idea of using Boolean functions encoded by means of binary decision diagrams (BDDs for short [Bry92]) to manipulate relations is due to Mac Millan & al (see for instance [BMDH90]). Since this pioneering paper, many works have been done on symbolic model checking, where transition systems are encoded by means of BDDs. Very impressive examples have been shown, demonstrating how powerful this approach is. BDDs have been used also to implement Boolean solvers of CLP languages [BS87].

With **Toupie**, we extend these ideas to obtain a full constraint language and thus we open the μ -calculus to a large spectrum of applications. Of course, problems that can be handled in this paradigm are of a different nature than those handled in CLP(\mathcal{FD}) (that come mainly from Operation Research). For instance, **Toupie** has been used to perform very efficient abstract interpretation of Prolog programs [CCMR93] and to verify mutual exclusion algorithms [CGR93].

In this paper, we show that **Toupie** is actually an efficient model-checker, or more precisely that the use of (extended) decision diagrams instead of binary ones (as done, for instance in [BMDH90, Bou93, EFT93]) improve the efficiency of symbolic model checking. We demonstrate “en passant” that the iterative squaring technique [Bry92], that seems a so pretty idea, is very doubtful in practice. We also show some funny issues in the computation of winning strategies in mathematical games.

It must be clear that these problems cannot be handled directly with the implemented solvers of CLP(\mathcal{FD}), due to the need of universal quantification and fixpoints.

The remaining of the paper is organized as follows: Section 2 is devoted to a presentation of the **Toupie** language. Sections 3 and 4 are devoted to applications. Finally, we examine the relation with other works in section 5.

2 The Constraint Language Toupie

2.1 Syntax and Semantics of Toupie Programs

A number of different versions of the propositional μ -calculus have been proposed in the literature. Hereafter, we summarize the syntax of **Toupie** programs, which departs, for many (technical) reasons, from the usual approaches.

There are two syntactic categories in **Toupie**: *formulae* and *predicate definitions*. A **Toupie** program is a set of predicate definitions, having different head predicate symbols. A **Toupie** query is a formula. Formulae have the following form:

- The two Boolean constants 0 and 1.
- $(X_1=X_2)$ or $(X_1=k)$ or $(X_1#X_2)$ or $(X_1#k)$ where X_1 and X_2 are variables and k is a constant symbol (# stands for disequality).
- $p(X_1, \dots, X_n)$ where p is an n -ary predicate variable and X_1, \dots, X_n are individual variables.
- $\neg f, f \wedge g, f \vee g, f \Leftrightarrow g, \dots$ where f and g are formulae and $\neg, \wedge, \vee, \Leftrightarrow$.
- **forall** $X_1, \dots, X_n f$ or **exist** $X_1, \dots, X_n f$ where X_1, \dots, X_n are variables and f is a formula.

Predicate definitions are as follows:

$p(X_1, \dots, X_n) += f$ or $p(X_1, \dots, X_n) -= f$ where p is an n -ary predicate variable, X_1, \dots, X_n are individual variables, and f is a formula. The tokens $+=$ and $-=$ denote respectively least and greatest fixpoint definition of the equation $P(X_1, \dots, X_n) = f$.

Each variable occurring in a fixpoint definition or request must have an interpretation domain. This domain must be declared with the first occurrence of the variable. A domain declaration is in the form: $X : \{ k_1, \dots, k_n \}$ or $X : i..j$ where X is a variable the k_i are constant symbols and i and j are integers (and thus $i..j$ denotes the corresponding range). It is possible to declare a default interpretation domain, and to name domains. In the following we denote by $dom(X)$ the interpretation domain of a variable X .

The semantics of **Toupie** programs is the attended one. That is that the fixpoint of an equation $p(X_1, \dots, X_n) = f$ is computed for the inclusion order in the powerset of $dom(X_1) \times \dots \times dom(X_n)$. The interested reader could refer to the appendix A for a precise denotational semantics. Note that the fixpoint definitions must be monotonic in order to ensure the existence of fixpoints and that this condition could be easily checked syntactically.

2.2 Decision Diagrams

Decision diagrams used in **Toupie** to encode relations, are an extension for symbolic finite domains of the binary decision diagrams [Bry92].

Shannon Decomposition of Relations

Definition 1. case connective

Let X be a variable, $dom(X) = \{k_1, \dots, k_r\}$ be its interpretation domain, and f_1, \dots, f_r be formulae. Then:

$$\text{case}(X, f_1, \dots, f_r) = ((X = k_1) \wedge f_1) \vee \dots \vee ((X = k_r) \wedge f_r)$$

Definition 2. Shannon Normal Form

A formula f is in Shannon normal form (SNF for short) if one of the following points holds:

- $f = 0$ or $f = 1$,
- $f = \text{case}(X, f_1, \dots, f_r)$, where X is a variable and $f_1 \dots f_r$ are formulae in SNF wherein X does not occur.

Property 3. Shannon Decomposition

Let $V = \{X_1, \dots, X_n\}$ be a set of variables, and $Const$ be a set of constants. Then, for any n -ary relation $R : (V \rightarrow Const) \rightarrow \mathcal{B}$ there exists a formula in SNF encoding R .

Reduced Ordered Decision Diagrams We first define decision diagrams:

Definition 4. Decision Diagrams

Let $V = \{X_1, \dots, X_n\}$ be a set of variables. A *Decision Diagrams* F is a directed acyclic graph such that:

- F has two leaves 0 and 1.
- Each internal node of F is labelled with a variable X belonging to V and if $dom(X) = \{k_1, \dots, k_r\}$ then the node has r outedges labelled with k_1, \dots, k_r .
- If a node labelled with the variable X is reachable from a node labelled with the variable Y then $X \neq Y$.

Now, it is clear that a decision diagram encodes a formula in SNF: the leaves encode the corresponding Boolean constants and each internal node encodes a *case* connective.

Now, we define a specific class of decision diagrams: reduced ordered decision diagrams.

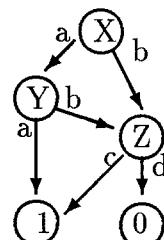
Definition 5. Reduced Ordered Decision Diagrams

Let $<$ be a total order over the variables X_1, \dots, X_n . A *Reduced Ordered Decision Diagram* F is a decision diagram such that:

- If a node labelled with the variable X is reachable from a node labelled with the variable Y then $X > Y$.
- Any node has at least two distinct sons ($case(X, f, \dots, f) \equiv f$).
- Two distinct nodes F and G are syntactically distinct, i.e. either they are labelled with different variables or there exists an index i such that the i -nth son of F is distinct of the i -nth son of G (reduction by means of maximum sharing of the sub-graphs)

In the remaining, we will consider only Reduced Ordered Decision Diagrams and call them Decision Diagrams (or DD for short).

Example 1. Let X, Y, Z be variables and $dom(X) = dom(Y) = \{a, b\}$ and $dom(Z) = \{c, d\}$. Let $p(X, Y, Z) = \{\langle a, a, c \rangle, \langle a, a, d \rangle, \langle a, b, c \rangle, \langle b, a, c \rangle, \langle b, b, c \rangle\}$; then the DD associated with p for the order $X < Y < Z$ is pictured beside. It encodes the formula: $case(X, case(Y, 1, case(Z, 1, 0)), case(Z, 1, 0))$ which is equivalent to p .



Property 6. Canonicity

Let R be a n -ary relation on the variables X_1, \dots, X_n and let $<$ be a total order over these variables. Then, there exists *one and only one* DD encoding R .

It follows that the test of equality between two relations encoded by means of two DDs is reduced to a test between the addresses of the DDs.

Logical Operations on DDs Decision Diagrams are also very efficient for performing logical operations on relations. The following property holds:

Property 7. Induction Principle

Let \odot be any binary logical operation and let $p = \text{case}(X, p_1, \dots, p_r)$ and $q = \text{case}(X, q_1, \dots, q_r)$ be two formulae in SNF. Then, the following equality holds:

$$\text{case}(X, p_1, \dots, p_r) \odot \text{case}(X, q_1, \dots, q_r) = \text{case}(X, p_1 \odot q_1, \dots, p_r \odot q_r)$$

It is easy to induce an effective procedure from this principle.

Memory Management for DDs Decision Diagrams encode relations over finite domains in a very compact way by means of the sharing of the subtrees. This sharing is automatically performed by storing the nodes in an hashtable: each time a node $\text{case}(X, p_1, \dots, p_r)$ is required, one first looks up the table and the node is created only if the node does not belong to it.

Another very important point that makes DDs efficient in practice is that the computation procedure uses a learning mechanism: each time a computation $p \odot q$ is performed, the result is memorized in an hashtable. Thus, this computation is never performed twice. Since the time required to an access in the hashtable is quasi-linear, the overhead due to this memorization is negligible. Moreover, the improvement obtained is often very big in practice, and becomes more and more important as the size of the problem grows up.

Variable Ordering Since the original paper by R. Bryant, it is well known that the size of a decision diagram (binary or not) crucially relies on the indices chosen for the variables. In his paper, R. Bryant gives an example where the BDD can be either linear or exponential w.r.t. the number of variables following the variable indexing.

By default, in **Toupie**, the variables are indexed with a very simple heuristic, known for its rather good accuracy. It consists in traversing the formula considered as a syntactic tree with a depth-first left-most procedure and to number variables in the induced order.

Nevertheless, this heuristic can produce very poor performances due to the projection operation. This operation is used each time a predicate $p(X_1, \dots, X_n)$ is called since the result of the computation of the corresponding fixpoint must be projected on the arguments of the call, here X_1, \dots, X_n . Projection can be dramatically unefficient if arguments are not ordered as the formal parameters.

This is the reason why, the user is allowed to define its own indices by $X@i$, where X is the first occurrence of a variable and i is any integer.

Advanced Features The effective implementation of fixpoint computations uses some tricky algorithms (projection by renaming and tabulation, dependency graphs) that avoid useless works and increase dramatically the performances. The interested reader could see [CR93] for a detailed presentation.

3 Symbolic Model Checking within Toupie

3.1 The Arnold-Nivat Model of Concurrency

The notion of *transition system* plays an important role for describing processes and systems of communicating processes. A simple way to represent processes widely used in many works on semantics and verification (*model checking*), is to consider that a process is a set of *states* and that an *action* or an *event* changes the current state of the process and can thus be represented as a transition between the two states. Transition systems are also used to describe systems of communicating processes: the states of the system are tuples of states of its components and the transitions are tuples of allowed transitions. The resulting automaton is called by Arnold and Nivat the synchronized product. This model of concurrency is the one used, for instance, in the model checker MEC [Arn89]. It is basically synchronous, even if it allows the description of non-synchronous phenomena.

The idea we use, first proposed by Mac Millan & al [BMDH90], is to encode transition systems in a symbolic way.

Individual Processes In order to illustrate this section, we model in **Toupie** the well-known Milner's scheduler [Mil89], a standard benchmark for process algebra tools [Bou93, EFT93]. The methodology remains the same for other problems such as the verification of mutual exclusion algorithms (see [CGR93]).

The scheduler consists of one starter process and N processes which are scheduled. The communication is organized in a ring. The transition system describing each cycler is depicted figure 1.

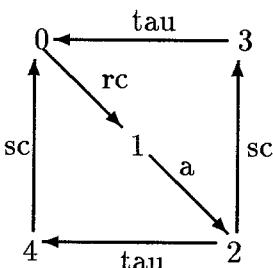


Fig. 1. The transition system encoding a cycler

Each cycler process C_i awaits the permit (rc) to start, performs the action a , and passes the turn (sc) to the next cycler either before or after some internal

computation (*tau*). The starter just initializes the process. A cycler is modeled in **Toupie** as follows:

```
let cycler_state 0..4      % definition of the domain "cycler_state"
let cycler_label {e,tau,a,sc,rc}

cycler(S:cycler_state,L:cycler_label,T:cycler_state) += (
  ((L=e) & (S=T))
  | ((S=0) & (L=rc) & (T=1)) | ((S=1) & (L=a) & (T=2))
  | ((S=2) & (L=sc) & (T=3)) | ((S=2) & (L=tau) & (T=4))
  | ((S=3) & (L=tau) & (T=0)) | ((S=4) & (L=sc) & (T=0))
)
```

The variables **S** and **T** stand for the sources and the targets of the transitions, the variable **L** stands for the labels of the transitions. Note that a transition labelled with **e** has been added. In the Arnold-Nivat Model, one considers that some action in some process can be executed only simultaneously with some other action in the other processes. In order to represent asynchronous actions, one adds transitions of the form $s \xrightarrow{e} s$, where **s** is a state and **e** is the label of the empty transition.

Synchronization Vector Now, one must synchronize the different processes, that is to constrain, for instance, the cycler *i* to emit a message (transition *sc*) when the cycler *i+1* receives the message (transition *rc*) and the other processes remain idle (transition **e**). The synchronization vector written as a **Toupie** rule is as follows:

```
synchronization_vector(
  SL:starter_label,C1L:cycler_label, ..., CnL:cycler_label) += (
    ((SL=sc) & (C1L=rc) & (C2L=e) & ... & (CnL=e))
    | ((SL=e) & (C1L=a) & (C2L=e) & ... & (CnL=e))
    | ((SL=e) & (C1L=tau) & (C2L=e) & ... & (CnL=e))
    | ((SL=e) & (C1L=sc) & (C2L=rc) & ... & (CnL=e))
    ...
)
```

Synchronized Product The computation of the synchronized product, that is the automaton modeling the behavior of the system of processes, can be now performed. The set of the reachable states of this product is computed as shown figure 2. It requires a fixpoint computation since the reachable states are found in a breadth-first way. The variable indices are not given in the figure. As remarked for instance in [EFT93] (for the Boolean case), the best order is the interleaved one, that is: $SS < LS < TS < C1S < C1L < C1T < \dots < CnS < CnL < CnT$. In the remaining we assume such an order.

3.2 Computing Properties

The predicates **reachable** and **edge** allow the verification of properties of the system:

```

edge(                                % allowed edges in the synchronized product
  SS:starter_state, ST:starter_state, % source and target in the starter
  C1S:cycler_state, C1T:cycler_state, % source and target in the cycler 1
  ...
  CnS:cycler_state, CnT:cycler_state) % source and target in the cycler n
+=
exist SL:starter_label, C1L:cycler_label, ... , C2L:cycler_label
(
  starter(SS,SL,ST)
  & cycler(C1S,C1L,C1T) & ... & cycler(CnS,CnL,CnT)
  & synchronization_vector(SL,C1L,...,C2L)
)

reachable(ST:starter_state,C1T:cycler_state, ..., CnT:cycler_state) += (
  initial_state(ST,C1T,...,CnT)
  | exist SS:starter_state, C1S:cycler_state, ... , C2S:cycler_state
    (reachable(SS,C1S,...,CnS) & edge(SS,ST,C1S,C1T,...,CnS,CnT))
)

```

Fig. 2. Reachable states in the synchronized product

Dead-Locks Let us recall that a dead-lock is a state wherein no transition is possible or only transitions leading to a deadlock state. The **Toupie** program to detect dead-locks is as follows:

$$\text{deadlock}(\mathbf{S}) += (\text{reachable}(\mathbf{S}) \wedge \forall \mathbf{T} (\text{reachable}(\mathbf{S}) \wedge \text{edge}(\mathbf{S}, \mathbf{T}) \Rightarrow \text{deadlock}(\mathbf{T}))$$

where **S** and **T** represent the variables ordered as previously.

Live-Locks The detection of live-locks is also a very important feature of a model checker. The problem arises when the modeled processes must share a critical ressource (a printer for instance). In this case, there is a live-lock in the system of processes if there is an infinite execution where:

- two processes attempt to access to their critical section, and never succeed
- none of the processes remains idle for ever.

The methodology consists in recomputing the set of reachable states by forbidding the states in which one process is in its critical section. There is no live-lock if and only if all the states of the obtained synchronized product are dead-locks.

Bisimulation A bisimulation is an equivalence relation between transition systems or different states of the same transition system (see the literature for a formal definition). The bisimulation generally considered on the Milner's scheduler is the observational equivalence that is to say that two states are equivalent

if and only if there is a path labeled with τ -transition joining them. This bisimulation is computed in **Toupie** in two steps: First, compute the τ -closure of the synchronized product, that is the paths of the form $\tau^* t \tau^*$, where t is any transition (predicate **tau_path**). Second, compute the equivalence relation between states using the extended edges above.

The second step is performed with a greatest fixpoint predicate:

```
equivalent(X, Y) == (
    reachable(X) ∧ reachable(Y)
    ∧ ∀L % L is a vector of labels, U and V are vectors of states
        ∀U tau_path(X, L, U) ⇒ ∃V (tau_path(Y, L, V) ∧ equivalent(U, V))
    ∧ ∀V tau_path(Y, L, V) ⇒ ∃U (tau_path(X, L, U) ∧ equivalent(U, V)))
```

The point is that the predicate **equivalent** mimics exactly the formal definition of the observational equivalence. Note also that if one wants to compute another bisimulation, it suffices to change the definition of the predicate **tau_path**.

Other Properties In [CGR93], we show also how the fairness and the safety of a mutual exclusion algorithm can be studied in **Toupie**. In mutual exclusion algorithm the fairness is achieved whenever the following fact holds:

- If a process P_i wants to access to its critical section, it succeeds in finite time. The safety is achieved if :
- Whenever a process P_i still remains in its non critical section (it does not attempt to reach critical section), then the mutual exclusion algorithm works.

All of these properties are expressed in **Toupie** in a very natural and declarative way.

Performances The table below indicates the running times for **Toupie** as well as those obtained by Bouali in the one hand [Bou93] and Enders & al in the other hand [EFT93] (the two last have been obtained on a SPARC 2 workstation, which is slightly faster than our own). These authors use BDDs based algorithms. The significant difference of performances in favour of **Toupie** comes, in our opinion, from the use of extended decision diagrams instead of binary ones. The interesting point is that very good performances can be obtained by using a general purpose constraint language instead of a specialized model checker.

processes	6	8	10	12	14	16	18	20
states	577	3073	15361	73729	$3 \cdot 10^6$	$1.2 \cdot 10^7$	$4.8 \cdot 10^7$	$1.8 \cdot 10^8$
transitions	2017	13825	84481	479233	$2 \cdot 10^7$	$8 \cdot 10^7$	$3.2 \cdot 10^8$	$1.28 \cdot 10^9$
reachable	0s70	1s30	2s03	3s36	4s41	5s76	7s36	9s36
Bouali	1s28	2s97	?	?	?	23s42	39s37	53s51
deadlock	0s10	0s13	0s18	0s21	0s26	0s30	0s38	0s38
bisimulation	4s08	6s70	9s95	14s23	18s01	23s20	28s40	34s76
Bouali	19s43	39s07	?	?	?	197s80	255s62	332s54
Enders & al	21s	40s	87s	145s	233s	348s	569s	850s

3.3 Iterative Squaring

A number of properties require to compute the transitive closure of the synchronized product, i.e. the pairs of global states $\langle \mathbf{S}, \mathbf{T} \rangle$ such that there exist a path from \mathbf{S} to \mathbf{T}

The transitive closure can be computed in two ways. First as follows:

$$\text{path}(\mathbf{S}, \mathbf{T}) + = \text{edge}(\mathbf{S}, \mathbf{T}) \vee \exists \mathbf{U} (\text{edge}(\mathbf{S}, \mathbf{U}) \wedge \text{path}(\mathbf{U}, \mathbf{T}))$$

Second, by means of the iterative squaring technique mentioned as a very powerful method by several authors:

$$\text{path}(\mathbf{S}, \mathbf{T}) + = \text{edge}(\mathbf{S}, \mathbf{T}) \vee \exists \mathbf{U} (\text{path}(\mathbf{S}, \mathbf{U}) \wedge \text{path}(\mathbf{U}, \mathbf{T}))$$

This technique is widely used, for instance for computing powers. Let \mathcal{K} be any ring and $X \in \mathcal{K}$ and $n \in \mathbb{N}$, then $X^{2n} = X \times X^{2n-1} = X^n \times X^n$. The second equality induces an iterative squaring method to compute a power.

Unfortunately, this pretty idea does not work for our purpose.

A critical example is the following: one considers N two states processes. At each step, one and only one of them changes of state. There is a single initial state. This example seems particularly in favour of the iterative squaring because:

- All the 2^N states of the free product are reachable.
- The number of iterations necessary to reach all the states is N while the number of squaring is $\log_2(N)$.

Surprisingly, it is not the case, as shown in the following table.

processes	4	8	16	32	64	128
path (iter.)	0s05	0s16	0s76	3s16	14s23	68s96
path (sqr.)	0s05	0s20	0s98	6s15	84s40	?

This phenomenon appears on almost all examples we have tried. A possible explanation could be that iterative squaring is efficient when the product of two objects has the same size than the objects themselves. Of course, it is not the case with DDs.

4 Winning Ways

4.1 Artificial Intelligence Classics

Toupie can be used to solve classical AI puzzles like N-Queens or Pigeon-Holes problems. Indeed, these problems do not require the expressiveness power of the language. Nevertheless, it is interesting to note that the performances are comparable with those obtained with CLP(\mathcal{FD}) languages based on the enumeration/propagation paradigm. The tests were performed on a Sparc 1 IPX, with 16MB RAM and 16MB of swap space. The running times given for CLP(\mathcal{FD}) have been obtained with (Cosytec) CHIP [Hen90] on a Sparc 2.

Queens The following table summarizes the running times for computing the Decision Diagrams that encode all the solutions of the N-queens problem for different values of N.

Queens	5	6	7	8	9	10
Toupie	0s05	0s06	0s25	0s58	2s20	6s73
CHIP	0s02	0s02	0s08	0s40	2s10	6s50

Pigeon-Hole The same for the pigeon-hole problem:

Pigeons/Holes	8/8	9/8	9/9	10/9	10/10	11/10	11/11	12/11	12/12	13/12
Toupie	0s05	0s45	0s25	1s25	2s73	3s11	10s31	9s75	25s63	34s40
CHIP	9s08	10s12	82s96	92s80	848s80	?	?	?	?	?

4.2 Games

More exciting is the analysis of two players mathematical games allowed in **Toupie**, thanks to the quantification and fixpoint mechanisms. The Nim game is a good illustration of this technics. Hereafter follows its rules:

The game begin with N lines numbered from 1 to N and containing $2 \times i - 1$ matches at line i . At each step, the player who has the turn takes as many matches as he wants in one of the line (but of course, at least one). Then the turn changes. The winner is the player who takes the last match.

In order to model the Nim game, one takes as many variables as there are lines (each variable taking its value in $0..2 \times i - 1$) plus one variable to model the turn. A move is represented by means of a predicate $move(\mathbf{S}, \mathbf{T})$ where the two vectors \mathbf{S} and \mathbf{T} encode two configurations of the variables (see section 3 for more explanations on the construction of this predicate).

The modeling of the configuration where there is a winning strategy is very simple and pretty in **Toupie**. It is programmed by means of two predicates:

$$\begin{aligned} winning(\mathbf{S}) &+ = \exists \mathbf{T} (move(\mathbf{S}, \mathbf{T}) \wedge losing(\mathbf{T})) \\ losing(\mathbf{S}) &+ = \forall \mathbf{T} (move(\mathbf{S}, \mathbf{T}) \Rightarrow winning(\mathbf{T})) \end{aligned}$$

Note that the player who has the turn loses when he (or she) cannot play any move. That is when his (or her) position (\mathbf{S}) is such that $\forall \mathbf{T} \neg move(\mathbf{S}, \mathbf{T})$ that is the initial step of the fixpoint computation.

The running times are reported in the following table.

lines	4	5	6	7	8
reachable configurations	384	3840	46080	645120	10321920
time to compute them	0s21	0s43	0s75	1s25	1s93
time to compute winning positions	0s50	1s73	6s23	25s18	141s60

5 Related Works

CLP(FD) As mentioned in the introduction, the nature of the problems handled with classical CLP(\mathcal{FD}) languages is different from the nature of the problems handled with **Toupi**e. This comes from the fact that the underlying data-structures are not the same. The use of DDs permits – from a practical point of view – the introduction of a full universal quantification and fixpoint computations but do not permit the use of branch and bound paradigm. There are strong motivations (thanks to applications) to introduce a **Toupi**e-like solver in a CLP(\mathcal{FD}) language complementarily to the currently implemented solvers. The introduction of universal quantification is rather simple. The introduction of a least fixpoint mechanism should not be too difficult by using tabulation mechanism, whilst the introduction of greatest fixpoints is more tedious.

Binary Decision Diagrams have been used in order to implement the Boolean solver of CHIP [BS87]. **Toupi**e can be seen as an extension of this work in several ways: extension of BDDs to finite domains, extension of the constraint language to the μ -calculus.

Model Checkers **Toupi**e is much more related to model checkers such as MEC [Arn89]. These programs are specialized for the verification of systems of finite state machines and communicating processes. Of course, they present the advantages and disadvantages of a specialized implementation: they are more efficient but far less flexible. It remains that DDs permit to encode in a very compact way even huge automata (DDs capture the regularity of these graphs by means of subtree sharing) — space consumption is the main problem of model checking — and that properties can be written in **Toupi**e in a very simple, elegant and declarative way.

Deductive Data Bases The semantics of **Toupi**e is close to the semantics of deductive data base languages. In particular, all the **Toupi**e formulae can be easily expressed in terms of the relational algebra. The difference comes, here again, from two points: first, in **Toupi**e the fixpoints definitions are explicitly declared, and can be either least or greatest fixpoints, with the possibility to mix both (under the condition that formulae remain monotonic). Note also that, since **Toupi**e semantics does not make the closed world assumption, the negation is naturally handled. Second, the underlying data structures are not the same. DDs allow a very efficient manipulation of relations, but are limited to small domains. Moreover, in the current implementation of **Toupi**e, all the created DDs are stored in memory and cannot be put on an external device.

6 Conclusion and Future Works

In this paper, we have presented several nontrivial applications of **Toupi**e. These applications show that μ -calculus over finite domains has a great expressive power and that this expressiveness is coupled with a good practical efficiency thanks to the use of Decision Diagrams.

Nevertheless Toupie can be improved in several ways: DD management, introduction of arithmetic builtins, heuristics for variable indexing, ...

Toupie can be considered from two different points of view:

– As a new solver for $\text{CLP}(\mathcal{FD})$ allowing a kind of relational calculus within this framework. This solver could come in addition to the classic ones. However, it remains some problems to integrate it smoothly (see section 5).

– As a new paradigm for constraint logic languages. In this case, the μ -calculus should be adapted in order to be a full programming language. This could be done in two ways: first restrict the language (for the constraint on the Herbrand universe) in order to make the relations computable by means of a tabulation mechanism. This implies to forbid general universal quantification and greatest fixpoints on this domain. Second, by using widening operators as proposed by the Cousot in [CC92]. This approach could be of a particular interest to analyse higher order functional languages as well as to introduce disjunction in constraints over continuous domains.

References

- [Arn89] A. Arnold. MEC: a System for Constructing and Analysing Transition Systems. In *Workshop on Automatic Verification Methods for Finite State Systems*, June 1989.
- [BMDH90] J.R. Burch, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *IEEE transactions on computers*, 1990.
- [Bou93] A. Bouali. *Etudes et mises en œuvre d'outils de vérification basée sur la bisimulation*. PhD thesis, Université Paris VII, 03 1993. in french.
- [Bry92] R. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys*, 1992.
- [BS87] W. Buettner and H. Simonis. Embedding Boolean Expressions into Logic Programming. *Journal of Symbolic Computation*, 4:191–205, 1987.
- [CC92] R. Cousot and P. Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. research report LIX/RR/92/09, Ecole Polytechnique, 1992.
- [CCMR93] M-M. Corsini, B. Le Charlier, K. Musumbu, and A. Rauzy. Efficient Abstract Interpretation of Prolog Programs by means of Constraint Solving over Finite Domains (extended abstract). In *Proceedings of the 5th Int. Symposium on Programming Language Implementation and Logic Programming, PLILP'93*, Estonia, 1993.
- [CGR93] M-M. Corsini, A. Griffault, and A. Rauzy. Yet another Application for Toupie: Verification of Mutual Exclusion Algorithms. In *proceedings of Logic Programming and Automated Reasonning, LPAR'93*. LNCS, 1993.
- [Col90] A. Colmerauer. An introduction to prologIII. *Communications of the ACM*, 28 (4), july 1990.
- [CR93] M-M. Corsini and A. Rauzy. First Experiments with Toupie. Technical Report 577–93, LaBRI - Université Bordeaux I, 1993.
- [EFT93] R. Enders, T. Filkorn, and D. Taubner. Generating BDDs for Symbolic Model Checking in CCS. *Journal of Distributed Computing*, 6:155–164, June 1993.

- [Hen90] P. Van Hentenryck. *Constraint Handling in Logic Programming*. Logic Programming. MIT Press, 1990.
- [JL87] J. Jaffar and J.L. Lassez. Constraint logic programming. In *Proceedings of Principle of Programming Languages (POPL'87)*, january 1987.
- [Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, New York, 1989.
- [Ull86] J. Ullman. Implementation of logical query languages for databases. *ACM Transactions on Database Systems*, 10, 03 1986.

A A Denotational Semantics of Toupie

The semantics of Toupie formulae is determined with respect to a structure $S = \langle \text{Const}, \mathcal{V} \rangle$ where Const is an interpretation domain, \mathcal{V} is a denumerable set of variables including all the variables of the program. As in DATALOG [Ull86], we assume 1) the unicity of names (two distinct constant symbols denotes two distinct constants) and 2) the closure of the domain, that is to say that Const is the set of all the constants occurring in the considered formula or program.

Definition 8. Individual Variable Assignments

An *individual variable assignment* is a mapping α from \mathcal{V} into Const such that $\alpha(X) \in \text{dom}(X)$ for all the variables X occurring in the program.

For sake of brevity, we assume in the following the condition $\alpha(X) \in \text{dom}(X)$.

Definition 9. Relations

A *relation* on S is a mapping from $\mathcal{V} \rightarrow \text{Const}$ into \mathcal{B} , where $X \rightarrow Y$ stands for the set of mappings from X to Y and \mathcal{B} stands for the Boolean values.

Definition 10. Predicate Variable Interpretations

Let Pr be the set of predicates occurring in the program. A *predicate variable interpretation* is a mapping from Pr into $(\mathbb{N} \rightarrow \text{Const}) \rightarrow \mathcal{B}$, where \mathbb{N} stands for the set of natural numbers.

This definition avoids the complications due to the different arities of the predicates. For a predicate of arity n , it suffices to consider that the corresponding function depends only on the first n numbers.

The semantics of a formula is thus a relation, and the semantics of a predicate (defined with a fixpoint equation) is a mapping from $(\mathbb{N} \rightarrow \text{Const})$ into \mathcal{B} .

A Toupie program P assigns a meaning to a set of predicate symbols Pr . The semantics of the program is defined as the least fixpoint of a transformation T . Let us note \mathcal{PR} the set $\text{Pr} \rightarrow (\mathbb{N} \rightarrow \text{Const}) \rightarrow \mathcal{B}$ of predicate variable interpretations. and \mathcal{RE} the set $(\mathcal{V} \rightarrow \text{Const}) \rightarrow \mathcal{B}$ of relations.

The program defines a continuous transformation:

$$T : \mathcal{PR} \rightarrow \mathcal{PR}$$

Each formula f defines a function:

$$T[f] : \mathcal{PR} \rightarrow \mathcal{RE}$$

And each equation defines a function:

$$T[\text{Eq}] : \mathcal{PR} \rightarrow (\mathbb{N} \rightarrow \text{Const}) \rightarrow \mathcal{B}$$

The definition of T will use the following notation.

Definition 11. Substitutions

Let $f : A \rightarrow B$ be a function. Let a_1, \dots, a_n be distinct elements of A and b_1, \dots, b_n be arbitrary elements of B . We note

$$f[a_1/b_1, \dots, a_n/b_n]$$

the function $g : A \rightarrow B$ such that $ga_i = fb_i$ ($1 \leq i \leq n$) and $ga = fa$ ($\forall a \notin \{a_1, \dots, a_n\}$).

The notation $[a_1/b_1, \dots, a_n/b_n]$ stands for $f[a_1/b_1, \dots, a_n/b_n]$ where f is an arbitrary function.

We are now in position to define the semantic function T . Let π be a predicate variable interpretation, α be an individual variable assignment, and σ be an element of $(\mathbb{N} \rightarrow \text{Const})$. T is defined inductively on the structure of formulae in the following way :

- $T[1] \pi \alpha = 1$ · and $T[0] \pi \alpha = 0$.
- $T[X_i = X_j] \pi \alpha = \alpha(X_i) = \alpha(X_j)$.
- $T[X_i = k] \pi \alpha = \alpha(X_i) = k$.
- $T[f \mid g] \pi \alpha = T[f] \pi \alpha \vee T[g] \pi \alpha$.
- $T[f \& g] \pi \alpha = T[f] \pi \alpha \wedge T[g] \pi \alpha$.
- $T[\forall X f] \pi \alpha = \bigwedge_{k \in \text{dom}(X)} (T[f] \pi \alpha[X/k])$.
- $T[\exists X f] \pi \alpha = \bigvee_{k \in \text{dom}(X)} (T[f] \pi \alpha[X/k])$.
- $T[P(X_{i_1}, \dots, X_{i_r})] \pi \alpha = \pi(P)(\alpha(X_{i_1}), \dots, \alpha(X_{i_r}))$.
- $T[P(X_1, \dots, X_n) + f] \pi \sigma = T[f] \pi [X_1/\sigma(1), \dots, X_n/\sigma(n)]$.
- Finally, the transformation associated with the program is:
 $T[Eq_1 \dots Eq_n] \pi = \pi[p_1/T[Eq_1] \pi, \dots, p_n/T[Eq_n] \pi]$
where the p_i are the predicates defined by the equations Eq_i .

Definition 12. Denotation of a Toupie Formula wrt a Program

Let P be a Toupie program. Let f be a Toupie formula. Let D be the set of variables occurring free in f . By definition, the *denotation* of f wrt P is the function $\mathcal{D}[f] : (D \rightarrow \text{Const}) \rightarrow \mathcal{B}$ such that, for all $\alpha \in (D \rightarrow \text{Const})$,

$$\mathcal{D}[f]\alpha = T[f](\mu(T[P]))\alpha',$$

where α' is any variable assignment such that

$$\alpha' X = \alpha X \ (\forall X \in D).$$

(The underlying program is kept implicit.)

Note that the introduction of least fixpoint definitions complicates the notations but not the semantics itself.

A Logical Denotational Semantics for Constraint Logic Programming

Alessandra Di Pierro * and Catuscia Palamidessi **

Abstract. The process interpretation of constraint logic programming (clp) leads to a model which is similar for many aspects to (an unsynchronized version of) concurrent constraint programming (ccp). However, it differs from the latter because it supports the notion of consistency: an action can be performed only if it does not lead to an inconsistent store. We develop a denotational, fully abstract semantics for clp with respect to successful, failed and infinite observables. This semantics extends the standard model of clp in two ways: on one hand by capturing infinite computations; on the other hand by characterizing a more general notion of negation. Finally, our work can be regarded as a first step towards the development of a simple model for ccp with atomic tell.

1 Introduction

Constraint logic programming (clp, [10]) is an extension of logic programming ([28]) in which the concept of unification on the Herbrand universe is replaced by the more general notion of constraint over an arbitrary domain. A program is a set of clauses possibly containing some constraints. A computation consists of a sequence of goals with constraints, where each goal is obtained from the previous one by replacing an atom by the body of a defining clause, and by adding the corresponding constraint, provided that consistency is preserved.

Like pure logic programming, clp has a natural computational model based on the so-called *process interpretation* ([27, 25]): the conjunction of atoms in a goal can be regarded as parallelism, and the selection of alternative clauses as nondeterminism. Such a model presents many similarities with the paradigm of concurrent constraint programming (ccp, [23]).

However there are some important differences between clp and ccp. The latter cannot be regarded just as clp plus concurrency mechanisms. A central aspect of clp, in fact, is that inconsistent computations (i.e. computations which lead to an inconsistent result) are eliminated as soon as the inconsistency is detected. More precisely, the mechanism of choice in clp embodies a *consistency check*: a branch, whose first action would add a constraint inconsistent with the store, is disregarded. Such a check is not supported in ccp. On the other hand, the choice of ccp is controlled by an *entailment check*, which allows to enforce synchronization among processes. Such synchronization mechanism is not present in clp.

* Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56125 Pisa, Italy. E.mail: dipierro@di.unipi.it

** DISI, Università di Genova, via Benedetto XV 3, 16132 Genova, Italy. E.mail: catuscia@di.unipi.it

The concern for consistency in clp is reflected also by the notion of observables: usually a distinction is made between *success* (existence of a computation which leads to a consistent result) and *failure* (all fair computations lead to inconsistent results). This distinction is not made in ccp: *false*, the inconsistent store, is regarded as a result having the same “status” as the other constraints.

In this paper we present a logical model for clp which extends its standard semantics (hence it maintains the distinction between success and failure), and, on the other hand, it captures the notions related to processes, like the “results” of infinite computations. We first consider a structured operational semantics, based on a transition system, by means of which we define the notion of observables as the set of final constraints resulting from all possible derivations. Then we develop a compositional semantics, based on logical operators, and show its full correspondence with the observables. It turns out that it is more convenient to reason about a generalized language, which we call Γ -clp, structured as the free language generated by a BNF grammar. Γ -clp subsumes clp; the main extension is that it allows the presence of global variables in the clause bodies.

Moreover, we use this model to treat the problem of negative goals. In logic programming the meaning of a negative goal $\neg G$ is based on the *finite failure* of G ([6]). This works only for ground goals and does not allow to define a notion of computed result. More refined approaches consider a constructive notion: the result of $\neg G$ is, roughly, the negation of the disjunction of all the possible results of G ([5, 29, 26]), or it is obtained by finding “fail answers” ([17, 7]). Our approach has some points in common with the latter, but it is based on a different philosophy: we treat negation operationally as any other construct of the language, by means of structural rules. Only in the observables we use a different definition, which takes into account the non-monotonic nature of negative goals.

Since our model characterizes also infinite computations, it allows us to define two notions of negation. One corresponds to the standard one, and considers only the finite results. The other considers also the results of infinite computations, and it captures the set of constraints which, when added by an hypothetical interactive process at some stage of the computation, will cause the computation to fail.

Unfortunately, due to the non-monotonic nature of negation, it is not possible to introduce negative goals into the language, unless some restrictions are made. In fact this would cause the loss of the continuity of the semantic operator which is used in the fixpoint construction of the denotational semantics. In the literature of logic programming one can find various proposals to solve this problem. In particular we cite two approaches based on syntactical conditions: stratification ([2]) and strictness ([13]). These ideas generalize smoothly to our case.

1.1 Related work

The problem of characterizing the infinite computations via a fixpoint (denotational, bottom-up, declarative ...) semantics has been extensively studied in (constraint) logic programming. The main challenge is to get such a characterization while maintaining a simple domain of denotations.

In most concurrent languages, for instance the imperative languages and the languages with global nondeterminism, the denotational characterization of the op-

erators requires complex structures, like synchronization trees or reactive sequences. On such domains there are well established techniques which allow to treat infinite computations, and they can be fruitfully applied also in the case of concurrent logic programming and concurrent constraint programming, see for instance [3], which is based on metric spaces. Another interesting approach, using category theory, is developed in [18].

In (constraint) logic programming, however, the domain of denotations for finite computations is particularly simple: sets of constraints or set of substitutions. Such a simple domain presents in principle more difficulties for treating infinite computations, because, for instance, it does not represent the occurrence of a computation step.

Most approaches aimed at modeling infinite computations with sets of constraints, are based on the greatest fixpoint of T_P , the immediate consequence operator which is used in logic programming for the fixpoint construction of the minimal model. Differences among these approaches depend on the kind of completion techniques applied on the underlying data structure, mainly based on partial orderings or metrics. However all these works have not been able to reach a full correspondence with the operational semantics. In the partial ordering completion of [8] only minimal answers are characterized. Furthermore the construction only works for clauses which contain at least one global variable. In the metric completion, at least in the approach found in the literature ([1, 14]) there is a basic soundness problem, because the objects which are considered are the solutions of the constraints rather than the constraint³ themselves, and it might be the case that an infinite element is the solution of a constraint whereas its finite approximations aren't. Hence a limit element obtained in the fixpoint construction might be unobtainable operationally.

A different approach, based on adding to the program some suitable clauses containing indefinite terms, and then applying a least fixpoint construction, has been developed in [15]. However, also in this case completeness is not achieved.

In [11] infinite computations have been studied from a declarative point of view. However, the purpose of that work is not to characterize the results, but rather to establish a distinction between *infinite successes* and *infinite failures*. An infinite computation is “successful” whenever all partial results of the computation allow the same solution (hence the limit result has a solution). Otherwise it is considered an infinite failure. Infinite successes are shown to correspond to the difference set between the greatest and the least fixed points of T_P . The others are the difference set between $T_P \downarrow \omega$ and the greatest fixpoint of T_P . In our model the second difference set disappears, because we work on completed domains which ensure the downward continuity of T_P ⁴. However also in our model a similar distinction between “successful” and “failed” infinite derivations can be made: infinite failures just correspond to the infinite computations delivering an inconsistent limit result.

The techniques we use in this paper have been inspired by the works in [12]

³ The language investigated in [1, 14] is pure logic programming, hence constraints are equalities over the Herbrand universe, and solutions are syntactical unifiers.

⁴ $T_P \downarrow \omega$ is the limit of the decreasing sequence $B, T_P(B), T_P^2(B), \dots$ where B is the Herbrand base (the top element of the domain). It can be shown that $T_P \downarrow \omega$ is the complement of the set of finite failures.

and [16], which present a semantics for (angelic) ccp based on Scott-compact sets, capturing also infinite behavior. The ideas behind the denotational construction are quite similar; the difference is that we deal with a language supporting a notion of consistency check.

As far as we know, our approach to negation is quite original.

1.2 Plan of the paper

In next section we recall the definition of constraint system. Section 3 gives a brief description of clp and introduces the language $\Gamma\text{-clp}$. Section 4 illustrates the operational semantics of $\Gamma\text{-clp}$ and the notion of observables. In Section 5 we develop the denotational model of $\Gamma\text{-clp}$ and show its full correspondence with the observables. Finally, in Section 6 we enrich the language with negative goals, and we study a generalized notion of negation. Due to lack of space, we omit the proofs; they can be found in the full paper.

2 Constraint System

The concept of constraints over arbitrary domain is central for the paradigm of clp and represents its major novelty with respect to logic programming. We follow here the approach of [22], which defines the notion of constraint system along the lines of Scott's *information systems* [24].

Roughly, an information system is based on a set of "propositions" with an entailment relation subject to a set of axioms. The *elements* of an information system are sets of propositions which are consistent and closed under entailment.

In [22] a constraint systems is defined as an information system, but for the requirement of the consistency, which is removed. This is necessary in ccp in order to capture the possibility that a program gives rise to an inconsistent state during its execution. In our case this would not be necessary. However, we maintain this extension because this approach leads to constraint systems which are complete algebraic lattices. Constraint systems having this property are very desirable domains since their powerdomains can be algebraically characterized in terms of some family of sets instead of a family of sets of sets ([20]). In Section 5 we will use this property to define a simple denotational semantics for the language $\Gamma\text{-clp}$.

In this paper we regard a constraint system as a complete algebraic lattice in which the ordering \sqsubseteq is the reverse of the entailment relation \vdash , the top element *false* represents the inconsistent constraint, the bottom element *true* the empty constraint, and the lub operation \sqcup the join of constraint, corresponding to the logical *and*. We refer to [22] for more details about the construction of such a structure.

Definition 1. A constraint system is a complete algebraic lattice $(\mathcal{C}, \sqsubseteq, \sqcup, \text{true}, \text{false})$ where \sqcup is the lub operation, and *true*, *false* are the least and greatest elements of \mathcal{C} , respectively.

An element $c \in \mathcal{C}$ is *compact*, or *finite*, iff for any directed subset D of \mathcal{C} , $c \sqsubseteq \sqcup D$ implies $c \sqsubseteq d$ for some $d \in D$. The lub of two finite elements is also finite.

Following the standard approach, we will sometimes use \vdash instead of \sqsubseteq . Formally $c \vdash d \Leftrightarrow d \sqsubseteq c$.

2.1 Cylindric Constraint Systems

In order to model local variables in $\Gamma\text{-clp}$, a sort of hiding operator is needed. This can be formalized by introducing a kind of constraint system which supports *cylindrification operators*, a notion borrowed from the theory of cylindric algebras, due to Henkin, Monk and Tarski ([9]).

Assume given a (denumerable) set of variables Var with typical elements x, y, z, \dots and consider a family of operators $\{\exists_x \mid x \in Var\}$. Starting from a constraint system C , construct a cylindric constraint system C' by taking $C' = C \cup \{\exists_x c \mid x \in Var, c \in C'\}$ modulo the identities and with the additional relations derived by the following axioms:

- (i) $\exists_x c \sqsubseteq c$,
- (ii) if $c \sqsubseteq d$ then $\exists_x c \sqsubseteq \exists_x d$,
- (iii) $\exists_x(c \sqcup \exists_x d) = \exists_x c \sqcup \exists_x d$,
- (iv) $\exists_x \exists_y c = \exists_y \exists_x c$,
- (v) if $\{c_i\}_i$ is an increasing chain, then $\exists_x \sqcup_i c_i = \sqcup_i \exists_x c_i$.

Note that these laws force \exists_x to behave as a first-order *existential operator*, as the notation suggests.

2.2 Diagonal constraint systems

In order to model parameter passing, it will be useful to enrich the constraint system with the so-called *diagonal constraints*, also from Henkin, Monk and Tarski ([9]).

Given a cylindric constraint system C' , define a diagonal constraint system C'' by taking $C'' = C' \cup \{d_{xy} \mid x, y \in Var\}$ modulo the identities and with the additional relations derived by the following axioms:

- (i) $d_{xx} = \text{true}$,
- (ii) if $z \neq x, y$ then $d_{xy} = \exists_z(d_{xz} \sqcup d_{zy})$,
- (iii) if $x \neq y$ then $c \sqsubseteq d_{xy} \sqcup \exists_x(c \sqcup d_{xy})$.

Intuitively, a constraint d_{xy} expresses the equality between x and y . Used together with the existential quantification these constraints allows us to model the variable renaming of a formula ϕ . In fact, thanks to the above axioms, the formula $\exists_x(d_{xy} \sqcup \phi)$ has precisely the meaning of ϕ with all the free occurrences of x replaced by y , i.e. $\phi[y/x]$.

3 The language

The syntax and the computational mechanism of clp ([10]) are very simple, but they do not provide a suitable basis to define a denotational semantics for clp. We need to reformulate the syntax of clauses and goals by means of a free grammar. The resulting language will be called $\Gamma\text{-clp}$. This will also allow us to describe the operational semantics in a structured way.

Constraints and atoms are basic constructs also in our language, but we restrict to atoms of the form $p(x)$. Furthermore we need to represent the conjunction of atoms,

the alternative choice among clauses, and locality. Correspondingly we introduce the operators \wedge , \vee and \exists_x . The choice of these symbols is because we have in mind a denotational semantics which assigns to goals a logical meaning, and the idea is that \wedge , \vee and \exists_x will correspond to the and, the or and the existential operator respectively. The \exists_x symbol here must not be confused with the analogous operator of the constraint system, but, of course, there is a close correspondence among them.

The grammar is described in Table 1. The language is parametric with respect to \mathcal{C} , and so is the semantic construction developed in this paper. We will assume in the following that there is at most one declaration for each predicate.

<i>Programs</i>	$P ::= \epsilon \mid D.P$
<i>Declarations</i>	$D ::= p(x) :- G$
<i>Goals</i>	$G ::= c \mid p(x) \mid G \wedge G \mid G \vee G \mid \exists_x G$

Table 1.: The language $\Gamma\text{-clp}$. The symbol p ranges over predicate names, and c ranges over the finite elements of a diagonal (and cylindric) constraint system \mathcal{C} .

Note that $\Gamma\text{-clp}$ subsumes clp. For instance a declaration for p consisting of two clauses

$$\begin{aligned} p(x, a) &:- c(x), q(x, y) \\ p(b, x) &:- \end{aligned}$$

can be rewritten in $\Gamma\text{-clp}$ as

$$\begin{aligned} p(z) &:- \exists_x (z = \langle x, a \rangle \wedge c(x) \wedge \exists_y \exists_w (w = \langle x, y \rangle \wedge q(w))) \\ &\quad \vee \\ &\quad \exists_x (z = \langle b, x \rangle). \end{aligned}$$

The language $\Gamma\text{-clp}$ is more general than clp for three reasons. First, it is not necessary to assume that \mathcal{C} contains the equality theory. Second, goals can contain disjunction and quantification. We like to have this feature because we think it provides the goals with a nice algebraic structure, which could be very useful, for instance, for developing a theory of equivalence. Third, in clp global variables can occur only in the goal, whereas in $\Gamma\text{-clp}$ they can occur also in the clauses.

4 Operational semantics

In this section we present the operational semantics of our language from a “process interpretation” point of view. Namely, we regard an atom in the goal as an agent, and a goal as a set of parallel agents which communicate with each other by establishing constraints on the global variables. In this view, a computation corresponds to the evolution of a dynamic network of parallel agents.

We define the operational semantics in the style of SOS ([19]), i.e. by means of a transition system which describes the evolution of the network in a structural way. The configurations are goals, and the transition relation, \longrightarrow , represents the computation step.

4.1 The problem of the consistency check

In order to avoid the generation of goals with inconsistent constraints, we have to perform an appropriate consistency check. In $\Gamma\text{-clp}$ it is more complicated than in clp, because of the generalized goals containing the \vee construct. We have to define what is the constraint associated to such goals, and how does it combine with the constraint of a parallel agent⁵. Intuitively, a goal of the form $G_1 \vee G_2$ will offer both the possibilities of G_1 and G_2 . If we put in parallel $G_1 \vee G_2$ with a goal G_3 we can avoid failure if and only if either G_1 or G_2 establish constraints which combine consistently with the ones of G_3 . In other words, we need a sort of logical *or*. A first idea would be to define the constraint associated to a disjunction as the *greatest lower bound* \sqcap of the two constraints of the disjuncts in the underlying constraint system. Unfortunately this choice does not work. Consider for example the constraint system illustrated in Figure 1. If we have the goal $x = 0 \vee x = 1$, then $x = 0 \sqcap x = 1 \equiv \text{true}$, which is consistent with the constraint $x = 2$. On the other hand, $(x = 0 \vee x = 1) \wedge x = 2$ should fail. The problem is that in order to correspond to the logical and and or, \sqcup and \sqcap should satisfy the distributive laws:

$$a \sqcup (b \sqcap c) = (a \sqcup b) \sqcap (a \sqcup c)$$

$$a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup (a \sqcap c).$$

This is not the case in this example: the lattice in Figure 1 is not distributive.

A possible solution is to embed the constraint system into a distributive one, where the lub and glb model conjunction and disjunction⁶. A simple way to do this is to lift to sets of constraints, following the idea presented in [4]. In fact the set union and the set intersection, which are the glb and the lub on sets, enjoy the distributive property. It turns out that we actually need to consider only sets which

⁵ An alternative would be to define the syntax and the operational semantics in such a way that disjunctions never occur in the goals. However this would complicate the transition system considerably.

⁶ Actually we do not need to have a complete lattice for dealing with the consistency check on the constraints generated during a computation. Since they are always finite, a lattice would be sufficient. However, we will need a complete lattice for the notion of observables (of infinite computations) and the denotational semantics.

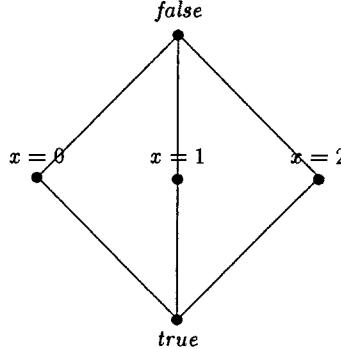


Fig. 1.: A non-distributive constraint system.

are *upward-closed* and *Scott-compact*. We recall that the upward closure of a set $D \subseteq \mathcal{C}$ is the set $\{c \in \mathcal{C} \mid \text{there exists } d \in D. d \sqsubseteq c\}$, which we denote by $\uparrow D$. A set $D \subseteq \mathcal{C}$ is upward-closed iff $D = \uparrow D$. Given a set $D \subseteq \mathcal{C}$, a *cover* for D is a set of compact elements C such that $D \subseteq \uparrow C$. Finally, a set $D \subseteq \mathcal{C}$ is Scott-compact iff every cover for D contains a finite subset which is also a cover. We will denote by $\mathcal{P}_{cu}(\mathcal{C})$ the set of Scott-compact upward-closed subsets of \mathcal{C} .

Given a cylindric, diagonal constraint system $(\mathcal{C}, \sqsubseteq, \sqcup, \text{true}, \text{false}, \text{Var}, \exists, d)$, consider the structure $\mathbf{C} = (\mathcal{P}_{cu}(\mathcal{C}), \supseteq, \cap, \text{True}, \text{False}, \text{Var}, \exists, \delta)$, where *True* is $\uparrow \text{true}$, which coincides with \mathcal{C} , and *False* is $\uparrow \text{false}$, which coincides with $\{\text{false}\}$. For $C \in \mathcal{P}_{cu}(\mathcal{C})$, $\exists_x C$ is defined as $\uparrow \exists_x C$, where $\exists_x C$ stands for the pointwise application of \exists_x to the elements of C . Finally, δ_{xy} is defined as $\uparrow d_{xy}$. We have the following property:

Proposition 2. \mathbf{C} is a cylindric, diagonal constraint system with finite lub and glb coinciding with set intersection and set union respectively. Furthermore, if $\{C_i\}_i$ is an increasing chain (w.r.t. \supseteq) then $\text{lub}_i C_i = \bigcap_i C_i$.

From the second part of this proposition it follows that

Corollary 3. The support lattice of \mathbf{C} is distributive.

The correspondence between the original constraint system and \mathbf{C} is obtained by mapping each element c into $\uparrow c$. It can be shown that this mapping preserves the ordering, the lub, and the existential operator. Of course, it does not preserve the glb.

The entailment relation on \mathbf{C} , i.e. the operational counterpart of the inverse of \supseteq , which we will denote by \Vdash , can be computed by extending the original entailment relation with the standard logical rules for conjunction and disjunction, plus

the axioms for the existential operators and the diagonal elements (\exists_x and δ_{xy}) corresponding to the laws of Sections 2.1 and 2.2.

However, our main concern is proving the consistency of constraint C , i.e. that $C \neq \text{False}$, or, equivalently, that $C \nmid\!\! \nmid \text{False}$ at least for those constraints C which belong to the set $\mathcal{L} \subseteq \mathcal{P}_{\text{cu}}(\mathcal{C})$ generated by the following grammar:

$$C ::= \uparrow c \mid C \cap C \mid \exists_x C \mid C \cup C.$$

To this purpose, having a complete deduction system for \Vdash is not sufficient, because it does not imply the computability of $\nmid\!\! \nmid$. Fortunately, in the case of the consistency check, we have a relatively complete system⁷ which is described in Table 2.

The idea behind this system is to reduce the expressions to disjunctions of elementary expressions of the form $\uparrow c$, whose consistency we are able to test directly on the underlying constraint system. Note that to make this reduction we use the distributivity of \cap wrt \cup and the properties $\exists_x \bigcup_i \uparrow c_i = \bigcup_i \uparrow \exists_x c_i$ and $\uparrow c \cap \uparrow d = \uparrow(c \sqcup d)$.

$\frac{}{\uparrow c \Vdash \uparrow d} c \vdash d$	$\frac{\bigcup_i \uparrow c_i \Vdash C, \bigcup_j \uparrow d_j \Vdash D}{\bigcup_{ij} \uparrow(c_i \sqcup d_j) \Vdash C \cap D}$
$\frac{\bigcup_i \uparrow c_i \Vdash C}{\bigcup_i \uparrow \exists_x c_i \Vdash \exists_x C}$	$\frac{\bigcup_i \uparrow c_i \Vdash C, \bigcup_j \uparrow d_j \Vdash D}{\bigcup_i \uparrow c_i \cup \bigcup_j \uparrow d_j \Vdash C \cup D}$
$\frac{}{\text{sat}(\uparrow c)} c \neq \text{false}$	$\frac{\exists j : \text{sat}(\uparrow c_j), \bigcup_i \uparrow c_i \Vdash C}{\text{sat}(C)}$

Table 2.: Deduction system for the consistency check. sat stands for “satisfiable”, i.e. consistent.

Proposition 4. (Relative completeness of sat) *The relation sat inductively defined by the system in Table 2 completely describes consistency in \mathcal{L} , i.e. for each element $C \in \mathcal{L}$, we derive $\text{sat}(C)$ iff $C \neq \text{False}$.*

Now we have the necessary machinery to define the operational semantics. Let's first define a function $\text{con} : \text{Goals} \rightarrow \mathcal{L}$, which gives the constraint established by a goal.

$$\begin{aligned} \text{Definition 5. } \quad \text{con}(c) &= \uparrow c \\ \text{con}(p(x)) &= \uparrow \text{true} \\ \text{con}(G_1 \wedge G_2) &= \text{con}(G_1) \cap \text{con}(G_2) \\ \text{con}(G_1 \vee G_2) &= \text{con}(G_1) \cup \text{con}(G_2) \\ \text{con}(\exists_x G) &= \exists_x \text{con}(G). \end{aligned}$$

⁷ Namely our system is complete provided that $c \neq \text{false}$ is semidecidable (hence decidable). This assumption is customary for the constraint systems used in clp.

The consistency check on a goal G consists in verifying $\text{sat}(\text{con}(G))$.

4.2 The transition system

The operational semantics of $\Gamma\text{-clp}$ is given by the transition relation \longrightarrow defined by the rules in Table 3. The program $P \equiv D_1.D_2.\dots.D_q$ is assumed to be fixed.

Recursion	$p(y) \longrightarrow \exists_\alpha(d_{y\alpha} \wedge \exists_x(d_{\alpha x} \wedge G)) \quad p(x) :- G \in P \text{ and } \text{sat}(\text{con}(G))$
Hiding	$\frac{G \longrightarrow G'}{\exists_x G \longrightarrow \exists_x G'}$
Disjunction	$\frac{G_1 \longrightarrow G'_1}{G_1 \vee G_2 \longrightarrow G'_1}$ $G_2 \vee G_1 \longrightarrow G'_1$
Parallelism	$\frac{\begin{array}{c} G_1 \longrightarrow G'_1 \\ G_1 \wedge G_2 \longrightarrow G'_1 \wedge G_2 \\ G_2 \wedge G_1 \longrightarrow G_2 \wedge G'_1 \end{array}}{\text{sat}(\text{con}(G'_1 \wedge G_2))}$

Table 3.: The transition system for $\Gamma\text{-clp}$.

The execution of a predicate call $p(y)$ is modeled by the recursion rule which replaces $p(y)$ by the body of its definition in the program P , after the link between the actual parameter y and the formal parameter x has been established. Following the method introduced in [23], we express this link by the context $\exists_\alpha(d_{y\alpha} \wedge \exists_x(d_{\alpha x} \wedge \dots))$, where α is a variable which does not occur in P . Note that through the whole computation only one variable α is needed. This mechanism for treating procedure calls is much simpler and more elegant than the machinery of standardization apart used in logic programming.

Disjunction is modeled by the arbitrary choice of one of the alternatives which do not bring to inconsistency. There is no need to write explicitly the consistency check, because the fact that G'_1 can be derived already guarantees its consistency. The same applies to the rule of hiding, in fact $\text{con}(G) \neq \text{False}$ implies $\text{con}(\exists_x G) \neq \text{False}$.

Parallel composition is modeled as interleaving.

Note that disjunction is the only rule which introduces a logical asymmetry between the antecedent and the subsequent of a computation step, in the sense that the “potential constraint” of one of the two disjuncts is discarded. This means that

the observables of a goal will have to be defined in terms of the collection of the results of all computations.

We will use the notations \longrightarrow^* to denote the reflexive and transitive closure of the transition relation \longrightarrow , and $\not\longrightarrow$ to indicate the absence of any further transition.

In the following, the class of “terminal” goals, namely those goals of the form $c_1 \wedge \dots \wedge c_n$ (i.e. consisting of constraints only) will be denoted by $TGoals$.

A computation is *and-fair* iff every goal which occurs in a \wedge -context either disappears sooner or later (because of an application of the disjunction rule) or it occurs as the premise in an application of the parallel rule.

4.3 The Observables

Following the standard definition, what we *observe* about a goal G in a program P is the set of constraints produced by its computations. The final constraints in case of terminating computations, the limits of intermediate constraints in case of infinite fair computations and *false* when all fair computations fail, namely they get stuck because the consistency check does not allow any further transition.

Definition 6. Given a program P , for every goal G we define

$$\begin{aligned} \mathcal{O}_P(G) = & \cup \{con(G') \mid G \longrightarrow^* G' \text{ for some } G' \in TGoals\} \\ & \cup \{ \uparrow false \mid \text{for every fair computation starting from } G, \\ & \quad G \longrightarrow^* G' \not\longrightarrow, \text{ for some } G' \notin TGoals\} \\ & \cup \{ \bigcap_n con(G_n) \mid \text{there exists an infinite fair computation} \\ & \quad G_0 = G \longrightarrow G_1 \longrightarrow \dots \longrightarrow G_n \longrightarrow \dots \}. \end{aligned}$$

Note that we consider a concept of *universal failure* according to the so-called notion of *don't know* nondeterminism: a failed computation branch is disregarded if there are successful computations.

5 Denotational semantics

Our aim here is to give a denotational characterization of the constraints computed by a goal.

As explained in the previous section, the constraint associated to a goal cannot be interpreted in \mathcal{C} : we need to consider a distributive structure. Hence the semantic function will map goals into $\mathcal{P}_{cu}(\mathcal{C})$. The elements of this domain will be called *processes*.

In order to treat predicate definitions and recursion, we need to introduce the notion of (semantic) environment, namely a function mapping predicate names into processes.

Definition 7. Let $Pred$ be a set of predicate symbols. Define

$$Env = \{e \mid e : Pred \rightarrow \mathcal{P}_{cu}(\mathcal{C})\},$$

with the ordering $e_1 \preceq e_2$ iff $\forall p. e_1(p) \supseteq e_2(p)$.

Since the ordering on Env is the pointwise extension of the ordering on a complete lattice, we have:

Proposition 8. *(Env, \preceq) is a complete lattice.*

The equations defining the semantic interpretation functions for the denotational semantics are defined in Table 4.

Programs	$\mathcal{D}[\epsilon]e = e$ $\mathcal{D}[D.P]e = \mathcal{D}[P](\mathcal{D}[D]e)$ $\mathcal{D}[p(x) :- G]e = e[\exists_x (\delta_{ax} \cap \mathcal{G}[G]e)/p]$
Goals	$\mathcal{G}[c]e = \uparrow c$ $\mathcal{G}[G_1 \wedge G_2]e = \mathcal{G}[G_1]e \cap \mathcal{G}[G_2]e$ $\mathcal{G}[\exists_x G]e = \exists_x \mathcal{G}[G]e$ $\mathcal{G}[G_1 \vee G_2]e = \mathcal{G}[G_1]e \cup \mathcal{G}[G_2]e$ $\mathcal{G}[p(y)]e = \exists_\alpha (\delta_{y\alpha} \cap e(p)) .$

Table 4.: The interpretation functions \mathcal{D} and \mathcal{G} .

Proposition 9. *For every program P and goal G the functions $\mathcal{D}[P] : \text{Env} \rightarrow \text{Env}$ and $\mathcal{G}[G] : \text{Env} \rightarrow \mathcal{P}_{\text{cu}}(\mathcal{C})$ defined in Table 4 are continuous.*

We define the meaning \mathcal{G}_P of a goal w.r.t. a program P , as

$$\mathcal{G}_P[G] = \mathcal{G}[G]\text{fix}(\mathcal{D}[P]),$$

where $\text{fix}(\mathcal{D}[P])$ is the least fixpoint of \mathcal{D} .

The observables \mathcal{O}_P and the semantic model \mathcal{G}_P for the $\Gamma\text{-clp}$ language given above are strictly related. In fact, they coincide.

Theorem 10. *For every program P and goal G , $\mathcal{G}_P[G] = \mathcal{O}_P(G)$ holds.*

6 A model for Negation

We consider now the possibility of introducing a construct for negation. We aim for the moment to have the possibility of computing negative goals, without using them in the bodies of the clause. Namely, in this work we restrict to *positive* programs.

Also we don't consider neither nested negation nor negation inside a \vee context. So, our goals are conjunctions of positive and negative goals. In the following, G stands for a positive goal as defined in previous sections.

First of all we have to define what is the constraint associated to a negated goal. More in general, we have to extend the structure C with a notion of negation. The first idea would be to define $\neg C$ as $C \setminus C$, but for doing this we should extend C with sets which are not upward closed. Another possibility is to define

$$\neg C = \{d \mid \forall c \in C. c \sqcup d = \text{false}\}.$$

This set is still upward closed, but might be not compact. It can be shown that the first notion of negation corresponds to classical negation, and the second one to intuitionistic negation. In this work we choose for the second, because it seems to combine better with the semantical construction developed so far. So, let's consider a structure C' which extends C in the sense that its support contains all the upward closed subsets of C , not only the Scott-compact ones. In this structure, the set union and set intersection are the glb and the lub operators also with respect to infinite sets (of sets). Furthermore, also the infinitary distributive laws hold.

From the point of view of the structural operational semantics, the evolution of $\neg G$ should be determined by the evolution of G . Hence we want to have a rule of the form

$$\text{Negation } \frac{G \longrightarrow G'}{\neg G \longrightarrow \neg G'} \quad \text{con}(\neg G') \neq \text{False}$$

However this rule, if combined with the notion of observables given before, is unsound. This is due to the asymmetry introduced by the disjunction rule: in general after an application of the disjunction rule we have $\text{con}(G') \subseteq \text{con}(G)$, therefore after the application of the negation rule we have that $\text{con}(\neg G')$ contains more constraints than the "original possibilities" of $\neg G$. Collecting the results like we did before would assign to negation a wrong meaning, as the following example shows.

Example 1. Consider the program

$$p(x) :- x = 0 \vee x = 1,$$

in a system where $true$, $x = 0$, $x = 1$ and $false$ are the only constraints. The goal $p(x)$ has two possible derivations:

$$p(x) \longrightarrow^* x = 0 \text{ and } p(x) \longrightarrow^* x = 1.$$

By the negation rule, $\neg p(x)$ has the derivations:

$$\neg p(x) \longrightarrow^* \neg(x = 0) \text{ and } \neg p(x) \longrightarrow^* \neg(x = 1).$$

Since $\neg(x = 0)$ corresponds to $\uparrow\{x = 1\}$ and $\neg(x = 1)$ corresponds to $\uparrow\{x = 0\}$, we would conclude that $p(x)$ and $\neg p(x)$ have the same observables!

However, the negation rule in itself is not unsound. It only makes necessary to adopt a suitable notion of observables.

6.1 Observing negation

As explained above, the negation rule reverts the asymmetry introduced by the disjunction rule. As a consequence, also the way we collect the observables must be dual: instead of the union, we have to take the intersection. We extend therefore the function \mathcal{O}_P on negative goals as follows:

Definition 11.

$$\begin{aligned} \mathcal{O}_P(\neg G) = & \cap \{con(\neg G') \mid \neg G \longrightarrow^* \neg G' \text{ for some } G' \in TGoals\} \\ & \cap \cap \{\uparrow true \mid \text{for every fair computation starting from } \neg G, \\ & \quad \neg G \longrightarrow^* \neg G' \not\longrightarrow, \text{ for some } G' \notin TGoals\} \\ & \cap \cap \{\bigcup_n con(\neg G_n) \mid \text{there exists an infinite fair computation} \\ & \quad \neg G_0 = \neg G \longrightarrow \neg G_1 \longrightarrow \dots \longrightarrow \neg G_n \longrightarrow \dots\}. \end{aligned}$$

This definition of observables does not correspond to the standard notion of *negation as finite failure* in (constraint) logic programming. To obtain such kind of negation we should include in $\mathcal{O}_P(\neg G)$ only the first two sets, corresponding to the information that we can derive from finite computations.

6.2 Denotation of negative goals

The denotational semantics \mathcal{G}_P extends to negative goals as follows:

$$\mathcal{G}_P[\neg G] = \neg \mathcal{G}_P[G].$$

The correspondence with the observables is maintained:

Proposition 12. $\mathcal{G}_P[\neg G] = \mathcal{O}_P(\neg G).$

Note that this definition of negation has a close correspondence with the set of the *finite failures* of G as defined in [21]:

$$\mathcal{O}_{P,fail}(G) = \{c \mid c \wedge G \longrightarrow^* \text{false}\}.$$

The difference is that in our case we have

$$\mathcal{O}_P(\neg G) = \{c \mid \mathcal{O}_P(c \wedge G) = \{\text{false}\}\},$$

which includes the possibility that G has an infinite computation giving *false* as the limit result.

7 Conclusions and future work

We have presented a generalized constraint logic programming with operators and, or, existential and (in a restricted form) negation. We have developed a structured operational semantics, embodying a mechanism for the appropriate consistency check. Then we have developed a natural model in which all the operators

of the language have a logical meaning. This model is actually a complete *Heyting algebra*, and therefore it should allow to define a notion of intuitionistic implication. A future objective is to investigate this feature to see whether its counterpart in the language would have significance.

Another topic which seems to be interesting is the development of a more sophisticated transition system for negation which would allow to achieve a sort of constructive negation. This could be done by manipulating negative goals so to simplify them: For instance, $\neg(G_1 \vee G_2)$ should be transformed into $\neg G_1 \wedge \neg G_2$, and $\neg(G_1 \wedge G_2)$ should be transformed into $\neg G_1 \vee \neg G_2$. This would also make more efficient the system, because it allows to drag disjunctions out of negation and apply the disjunction rule. Another advantage is that in this way we don't need anymore to define a different notion of observables: the negative goals will in fact be completely reduced, at the end of a computation, to conjunctions of constraints. Therefore, we would have a structural semantics even when negation is a free constructor in the goals, thus generating nested negations, conjunctions of negations etc.

Another problem to investigate is how to allow negations in the bodies of the clauses. As explained in the introduction, this would compromise the monotonicity and the continuity of the semantic operator \mathcal{D} , which therefore would not be guaranteed to have a least fixpoint. So, either we put some restrictions on the clauses with negation, like stratification ([2]) or and strictness ([13]), or we find another way (some appropriate fixpoint) to characterize the intended meaning of a program.

References

1. M.A. Nait Abdallah. On the interpretation of infinite computations in logic programming. In J. Paredaens, editor, *Proc. of Automata, Languages and Programming*, volume 172, pages 374–381. Springer Verlag, 1984.
2. K. R. Apt, H. Blair, and A. Walker. Towards a Theory of Declarative Knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, Los Altos, Ca., 1988.
3. F.S. de Boer, J.N. Kok, C. Palamidessi, and J.J.M.M. Rutten. Semantic models for Concurrent Logic Languages. *Theoretical Computer Science*, 86(1), 3–33, 1991.
4. F.S. de Boer, M. Gabbrielli, E. Marchiori, and C. Palamidessi. Proving Concurrent Constraint Programs Correct. In *Proc. Eighteenth Annual ACM Symp. on Principles of Programming Languages*, 1993.
5. D. Chan. Constructive Negation Based on the Completed Database. In R. A. Kowalski and K. A. Bowen, editors, *Proc. Fifth Int'l Conf. on Logic Programming*, pages 111–125. The MIT Press, 1988.
6. K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.
7. W. Drabent. Constructive Negation by Fail Answers. In *Proc. of the Workshop on Logic Programming and Non-monotonic reasoning*, 1993. To appear.
8. W.G. Golson. Toward a declarative semantics for infinite objects in logic programming. *Journal of Logic Programming*, 5:151–164, 1988.
9. L. Henkin, J.D. Monk, and A. Tarski. *Cylindric Algebras (Part I)*. North-Holland, 1971.
10. J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. Fourteenth Annual ACM Symp. on Principles of Programming Languages*, pages 111–119, 1987.

11. J. Jaffar and J.-L. Lassez. Constraint Logic Programming. Technical report, Department of Computer Science, Monash University, June 1986.
12. R. Jagadeesan, V.A. Saraswat, and V. Shanbhogue. Angelic non-determinism in concurrent constraint programming. Technical report, Xerox Park, 1991.
13. K. Kunen. Signed Data Dependencies in Logic Programs. *Journal of Logic Programming*, 7(3):231–245, 1989.
14. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987. Second edition.
15. G. Levi and C. Palamidessi. Contributions to the semantics of logic perpetual processes. *Acta Informatica*, 25(6):691–711, 1988.
16. M. Z. Kwiatkowska. Infinite Behaviour and Fairness in Concurrent Constraint Programming. In J. W. de Bakker, W. P.de Roever, and G. Rozenberg, editors, *Semantics: Foundations and Applications*, volume 666 of *Lecture Notes in Computer Science*, pages 348–383, Beekbergen The Nederland, June 1992. REX Workshop, Springer-Verlag, Berlin.
17. J. Maluszyński and T. Näslund. Fail Substitutions for Negation as Failure. In E. Lusk and R. Overbeck, editors, *Proc. North American Conf. on Logic Programming'89*, pages 461–476. The MIT Press, 1989.
18. S. Nystrom and B. Jonsson. In D. Miller, editor, *Proc. International Symposium on Logic Programming'93*, pages 335–352. The MIT Press, 1993.
19. G. Plotkin. A structured approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.
20. G. Plotkin. Domains. Department of Computer Science, University of Edinburgh, 1992. Post-graduate lecture notes in advanced domain theory (incorporating the 'Pisa notes' 1981).
21. J. C. Shepherdson. A sound and complete semantics for a version of negation as failure. *Theoretical Computer Science*, 65:343–371, 1989.
22. V.A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proc. Seventeenth Annual ACM Symp. on Principles of Programming Languages*, pages 232–245, New York, 1990.
23. V.A. Saraswat, M. Rinard, and P. Panangaden. Semantics foundations of concurrent constraint programming. In *Proc. Eighteenth Annual ACM Symp. on Principles of Programming Languages*, New York, 1991.
24. D. Scott. Domains for denotational semantics. In *Proc. of ICALP*, 1982.
25. E.Y. Shapiro. A subset of Concurrent Prolog and its interpreter. Technical Report TR-003, Institute for New Generation Computer Technology, Tokyo, 1983.
26. P. Stuckey. Constructive negation for constraint logic programming. In *Proc. sixth Annual Symposium on Logic in Computer Science*, 1991.
27. M. H. van Emden and G. J. de Lucena. Predicate logic as language for parallel programming. In K. L. Clark and S. A. Tä rnlund, editors, *Logic Programming*, pages 189–198. Academic Press, London, 1982.
28. M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of ACM*, 23(4):733–742, 1976.
29. M. G. Wallace. Negation by Constraints: a Sound and Efficient Implementation of Negation in Deductive Databases. In *IEEE Int'l Symp. on Logic Programming*, pages 253–263. IEEE, 1987.

Compilation of Head and Strong Reduction

Pascal Fradet

INRIA/IRISA

Campus de Beaulieu, 35042 Rennes Cedex, France

fradet@irisa.fr

Abstract

Functional language compilers implement only weak-head reduction. However, there are cases where head normal forms or full normal forms are needed. Here, we study how to use cps conversion for the compilation of head and strong reductions. We apply cps expressions to a special continuation so that their head or strong normal form can be obtained by the usual weak-head reduction. We remain within the functional framework and no special abstract machine is needed. Used as a preliminary step our method allows a standard compiler to evaluate under λ 's.

1 Introduction

Functional language compilers consider only weak-head reduction and the evaluation stops when a weak head normal form (whnf), that is a constant or a λ -abstraction, is reached. In practice, whnf's are considered sufficient because printable results belong to basic domains. However, there are cases where one would like to reduce under λ 's to get head normal forms (hnf) or even (strong) normal forms (nf). Specifically, head/strong reduction can be of interest in:

- program transformations (like partial evaluation) which need to reduce under λ 's,
- higher order logic programming like λ -prolog [15] where unification involves reducing λ -terms to normal forms,
- evaluating data structures coded in λ -expressions,
- compiling more efficient evaluation strategies.

A well known tool used to compile (weak) evaluation strategies of functional programs is continuation-passing style (cps) conversion [6][16]. This program transformation makes the evaluation ordering explicit. We see it as a compiling tool since cps expressions can be reduced without any dynamic search for the next redex. Its main advantage is that it stays within the functional framework and thus does not preclude further transformations. Several compilers for strict and non-strict functional languages integrate a cps conversion as a preliminary step [1][7][11].

Here, we study how to use cps conversion for the implementation of head and strong reductions. To the best of our knowledge, the application of this transformation to such reduction strategies has not been investigated for far. A key property of cps expressions is that their (weak) evaluation is order independent: there is a unique (weak) redex at each reduction step. This property does not hold with strong or head reduction ; a cps expression may have several (strong) redexes. Our approach is to simulate head/strong reductions by weak reductions. Cps expressions are applied to special continuations so that their head/strong normal form can be obtained by the usual weak-head reduction. This way, we still use the only strategy known by compilers (weak reduction), and we retain the key property of cps. The advantage of this approach is that we do not have to introduce a special abstract machine and/or particular structures. It can be used to extend an existing compiler with head/strong reduction capabilities and it enables us to use classical implementation and optimization techniques.

In the following, we assume a basic familiarity with the λ -calculus and cps. In section 2, we introduce some notations, the definitions of the different reduction strategies and cps conversion. We consider in section 3 how to use standard cps conversion to simulate head-reduction of λ -expressions. Section 4 is devoted to strong reduction which involves a minor modification of the technique used for head reduction. In section 5, we envisage a restriction of λ -calculus with a flexible notion of typing which allows a better treatment of head reduction. Section 6 describes how this method could be used to compile more efficient reduction strategies, addresses implementation issues and discusses possible extensions.

2 Preliminaries

One of the application of head reduction being to avoid duplicated or useless computations (see section 6), we will focus on call-by-name. We consider pure λ -calculus and the global λ -expression to be reduced is always assumed to be closed. Given a reduction strategy x , $E \xrightarrow{x} F$ (resp. $E \xrightarrow{i} F$) reads “E reduces to F after one (resp. i) reduction step by x”. The transitive, reflexive closure of \xrightarrow{x} is noted $\xrightarrow{*x}$. The three computation rules we are dealing with (i.e. weak head, head and strong reduction) are described in the form of deductive systems.

- Weak head reduction is noted \xrightarrow{w} and is defined by

$$(\lambda x.E) F \xrightarrow{w} E[F/x] \quad \frac{E \xrightarrow{w} E'}{E F \xrightarrow{w} E' F}$$

Closed whnf's are of the form $\lambda x.E$.

- Head reduction is noted \xrightarrow{h} and is defined by

$$\frac{E \xrightarrow{w} E'}{\lambda x_1. \dots \lambda x_n. E \xrightarrow{h} \lambda x_1. \dots \lambda x_n. E'} \quad n \geq 0$$

Closed hnf's are of the form $\lambda x_1. \dots \lambda x_n. x_i E_1 \dots E_p$ ($1 \leq i \leq n$, $p \geq 0$). x_i is called the head variable.

- Strong reduction is noted \xrightarrow{s} and, with N_i 's standing for normal forms, is defined by

$$\frac{E \xrightarrow{h} E'}{E \xrightarrow{s} E'} \quad \frac{E_i \xrightarrow{s} E'_i}{\lambda x_1. \dots \lambda x_n. x_i E_1 \dots E_i N_{i+1} \dots N_p \xrightarrow{s} \lambda x_1. \dots \lambda x_n. x_i E_1 \dots E'_i N_{i+1} \dots N_p}$$

Strong reduction is described as a sequence of head reductions. When a hnf is reached, the arguments of the head variable are reduced in a right to left fashion. Closed normal forms are of the form $\lambda x_1. \dots \lambda x_n. x_i N_1 \dots N_p$ with $1 \leq i \leq n$ and with $N = x_j \mid \lambda x_1. \dots \lambda x_n. x_k N_1 \dots N_p$

\mathcal{N} stands for the standard cps conversion associated with call-by-name [16] and is defined in Figure 1. Call-by-name cps could also been defined *à la* Fischer where continuations occur first [6]. Our approach could be applied to this kind of cps expressions as well.

$$\mathcal{N}(x) = x$$

$$\mathcal{N}(\lambda x.E) = \lambda c.c (\lambda x.\mathcal{N}(E))$$

$$\mathcal{N}(E F) = \lambda c.\mathcal{N}(E) (\lambda f.f \mathcal{N}(F) c)$$

Figure 1 Standard Call-by-Name Cps

Variables c and f are supposed not to occur in the source term. The reduction of a cps term consists of a sequence of administrative reductions (i.e. reduction of redexes introduced by the transformation, here redexes of the form $(\lambda c.E) F$ or $(\lambda f.E) F$), followed by a proper reduction

(corresponding to a reduction of the source term), followed by administrative reductions, and so on. The relation induced by administrative reductions is noted \xrightarrow{A} , for example:

$$\mathcal{N}(\lambda x.E) F \xrightarrow{I} (\lambda c.(\lambda c.c (\lambda x.\mathcal{N}(E))) (\lambda f.f \mathcal{N}(F) c)) I \xrightarrow{A} (\lambda x.\mathcal{N}(E)) \mathcal{N}(F) I$$

The following property states that evaluation of cps expressions simulates the reduction of source expressions ; it is proved in [16].

Property 1 If $E \xrightarrow{*} W$ then $\mathcal{N}(E) I \xrightarrow{*} X \xleftarrow{W} \mathcal{N}(W) I$ and if W is a whnf then X is a whnf. Furthermore E does not have a whnf iff $\mathcal{N}(E) I$ does not have a whnf.

Cps conversion introduces many new λ -abstractions and affects the readability of expressions. In the remainder of the paper we use the following abbreviations

$$\lambda_c x.E \equiv \lambda c.c (\lambda x.E)$$

$$\lambda_c \vec{x}_n . E \equiv \lambda c.c (\lambda x_1. \dots (\lambda c.c (\lambda x_n.E)) \dots)$$

$$\vec{X}_n . E \equiv \lambda f.f X_1 (\dots (\lambda f.f X_n E) \dots)$$

3 Head Reduction

Since we are interested in compiling, we consider only programs, i.e. closed expressions. A compiler does not know how to deal with free variables ; the expression to be reduced must remain closed throughout the evaluation. Furthermore, in order to use weak head reduction to evaluate hnf's, the leading λ 's must be suppressed as soon as the whnf is reached. Our solution is to apply the whnf to combinators so that the associated variables are replaced with closed expressions. The head lambdas disappear, the expression remains closed and the evaluation can continue as before. After the body is reduced to hnf the expression must be reconstructed (i.e. the leading λ 's must be reintroduced as well as their variables). We reach a hnf when the head variable is applied (a closed hnf is of the form $\lambda x_1. \dots \lambda x_n.x_i E_1 \dots E_n$) so the combinators previously substituted for the leading variables should take care of the reconstruction process.

In general, it is not possible to know statically the number of leading λ 's (sometimes called the binder length) of the hnf of an expression. We have to keep track of their number in order to eventually reintroduce them. This complicates the evaluation and reconstruction process. In section 5 we present a means of avoiding this need for counting.

We use the standard call-by-name cps conversion (\mathcal{N}). The global cps expression is applied to a recursive continuation Ω and an index n such that $\Omega E n = E H_n \Omega n+1$ (Ω , H_n and n being combinators). Combinators n represent the number of head abstractions already encountered. The weak head reduction of such expressions looks like

$$\mathcal{N}(E) \Omega \bar{n} \xrightarrow{*} (\lambda c.c (\lambda x.F)) \Omega \bar{n}$$

when a cps expression E is evaluated by wh-reduction, its whnf (if any) will be of the form $\lambda c.c (\lambda x.F)$

$$\xrightarrow{w} \Omega (\lambda x.F) \bar{n}$$

the continuation Ω is applied

$$\xrightarrow{w} (\lambda x.F) H_n \Omega \bar{n+1}$$

Ω applies the whnf to combinator H_n , Ω and the new index

$$\xrightarrow{w} F[H_n/x] \Omega \bar{n+1}$$

H_n is substituted for x

The expression remains closed and the evaluation continues, performing the same steps if other whnf's are encountered. Eventually a hnf is reached, that is, a combinator H_i is in head position and this combinator is responsible for reconstructing the expression.

In fact, we do not apply the global expression directly to Ω but to combinator A (defined by $A E = F E$) whose task is to apply the expression to Ω . This way Ω remains outside the expression and it makes its suppression during the reconstruction process easier. This technical trick is not absolutely necessary but it simplifies things when working within the pure λ -calculus. The reduction steps that occur when a whnf is reached actually are

$$(\lambda c.c (\lambda x.F)) A \Omega \bar{n} \xrightarrow{w} A (\lambda x.F) \Omega \bar{n} \xrightarrow{w} \Omega (\lambda x.F) \bar{n} \xrightarrow{w} (\lambda x.F) H_n A \Omega \bar{n+1}$$

If E is a closed expression, its transformed form will be $\mathcal{N}(E) A \Omega \bar{0}$ with

$$A M N \xrightarrow{w} N M \quad (A)$$

$$\Omega M \bar{n} \xrightarrow{w} M H_n A \Omega \bar{n+1} \quad (\Omega)$$

The family of combinators H_i is defined by

$$H_i M N \bar{n} = \lambda_c \vec{x}_n . \lambda_{c,x_{i+1}} (M (R \bar{n} (\lambda_c. \vec{x}_n . c)) (K c)) \quad (H)$$

with $R E F G H I \xrightarrow{w} \lambda f.f (\lambda c.G A \Omega E (F c)) (H (R E F) I)$ (R)

The definitions (H) and (R) can be explained intuitively as follows. When the hnf is reached the expression is of the form $H_i (\lambda f.f E_1 \dots (\lambda f.f E_m A) \dots) \Omega n$, n representing the number of head abstractions of the hnf. The reduction rule of H_i deletes Ω , reintroduces the n leading λ 's, the head variable and yields

$$\lambda_c \vec{x}_n . \lambda_{c,x_{i+1}} ((\lambda f.f E_1 \dots (\lambda f.f E_m A) \dots) (R \bar{n} (\lambda_c. \vec{x}_n . c)) (K c))$$

Some H_i 's may remain in the continuation of x_{i+1} and the role of R is to remove them by applying each E_i to suitable arguments. The reconstructing expression $R n (\lambda_c. \vec{x}_n . c)$ will be recursively called by the argument "list" $(\lambda f.f E_1 \dots (\lambda f.f E_m A) \dots)$; the final continuation A will call K which removes $R n (\lambda_c. \vec{x}_n . c)$. Meanwhile R applies each argument E_i to $A, \Omega, n, (\vec{x}_n . c)$ and reconstructs the argument "list". In summary

$$(\lambda f.f E_1 \dots (\lambda f.f E_m A) \dots) (R \bar{n} (\lambda_c. \vec{x}_n . c)) (K c) \xrightarrow{w} (\lambda f.f (\lambda c.E_1 A \Omega \bar{n} (\vec{x}_n . c)) \dots \\ \dots (\lambda f.f (\lambda c.E_m A \Omega \bar{n} (\vec{x}_n . c)) c) \dots)$$

Each E_i corresponds to an original cps expression F_i containing at most n free variables such that $E_i \equiv F_i[\vec{x}_n / \vec{H}_n]$. Let N_i be the normal form of F_i then

$$\lambda c. E_i A \Omega \bar{n} (\vec{x}_n . c) = \lambda c. F_i[\vec{x}_n / \vec{H}_n] A \Omega \bar{n} (\vec{x}_n . c) = \lambda c. (\lambda_c \vec{x}_n . F_i) A \Omega \bar{0} (\vec{x}_n . c)$$

(and using Property 3) $= \lambda c. (\lambda_c \vec{x}_n . N_i) (\vec{x}_n . c) = \lambda c. N_i c = N_i$

So, the reduction of $\lambda c. E_i A \Omega \bar{n} (\vec{x}_n . c)$ eventually yields the normal form of the argument, suppressing this way the combinators H_i 's occurring in E_i .

Example: Let $E \equiv \lambda x. (\lambda w. \lambda y. w y x) (\lambda z. z) x$

Its head reduction is

$$E \xrightarrow{h} \lambda x. (\lambda y. (\lambda z. z) y x) x$$

$$\xrightarrow{h} \lambda x. (\lambda z. z) x x$$

$$\xrightarrow{h} \lambda x. x x$$

After cps conversion and simplification the expression becomes

$$\mathcal{N}(E) = \lambda_c x. \lambda c. (\lambda w. \lambda_c y. \lambda c. w (\lambda f. f y (\lambda f. f x c))) (\lambda_c z. z) (\lambda f. f x c)$$

The weak head reduction of $\mathcal{N}(E)$ $A \Omega \bar{0}$ simulates the head reduction of E . Reductions corresponding to head reductions of the source expression are marked by \ddagger ; the other being administrative reductions.

$$\begin{aligned}
 \mathcal{N}(E) A \Omega \bar{0} &\xrightarrow[w]{} A (\lambda x. \lambda c. (\lambda w. \lambda_c y. \lambda c. w (\lambda f. f y (\lambda f. f x c))) (\lambda_c z. z) (\lambda f. f x c)) \Omega \bar{0} \\
 &\xrightarrow[w]{} \Omega (\lambda x. \lambda c. (\lambda w. \lambda_c y. \lambda c. w (\lambda f. f y (\lambda f. f x c))) (\lambda_c z. z) (\lambda f. f x c)) \bar{0} \\
 &\xrightarrow[w]{} (\lambda x. \lambda c. (\lambda w. \lambda_c y. \lambda c. w (\lambda f. f y (\lambda f. f x c))) (\lambda_c z. z) (\lambda f. f x c)) H_0 A \Omega \bar{1} \\
 &\xrightarrow[w]{} (\lambda w. \lambda_c y. \lambda c. w (\lambda f. f y (\lambda f. f H_0 c))) (\lambda_c z. z) (\lambda f. f H_0 A) \Omega \bar{1} \\
 &\xrightarrow[w]{} (\lambda_c y. \lambda c. (\lambda_c z. z) (\lambda f. f y (\lambda f. f H_0 c))) (\lambda f. f H_0 A) \Omega \bar{1} \quad \ddagger \\
 &\xrightarrow[w]{} (\lambda y. \lambda c. (\lambda_c z. z) (\lambda f. f y (\lambda f. f H_0 c))) H_0 A \Omega \bar{1} \\
 &\xrightarrow[w]{} (\lambda c. (\lambda_c z. z) (\lambda f. f H_0 (\lambda f. f H_0 c))) A \Omega \bar{1} \quad \ddagger \\
 &\xrightarrow[w]{} (\lambda z. z) H_0 (\lambda f. f H_0 A) \Omega \bar{1} \\
 &\xrightarrow[w]{} H_0 (\lambda f. f H_0 A) \Omega \bar{1} \quad \ddagger
 \end{aligned}$$

The hnf is reached. Using the definition of H_0 we get

$$H_0 (\lambda f. f H_0 A) \Omega \bar{1} \rightarrow \lambda_c x. \lambda c. x ((\lambda f. f H_0 A) (R \bar{1} (\lambda c. \lambda f. f x c)) (K c)) \equiv \Delta \quad (H)$$

Now, we show that this hnf Δ is equivalent to (or that the reconstruction yields) the principal hnf $(\lambda x. x x)$ in cps form $(\lambda_c x. \lambda c. x (\lambda f. f x c))$.

$$\begin{aligned}
 \Delta &\rightarrow \lambda_c x. \lambda c. x (R \bar{1} (\lambda c. \lambda f. f x c) H_0 A (K c)) \\
 &\rightarrow \lambda_c x. \lambda c. x (\lambda f. f (\lambda c. H_0 A \Omega \bar{1} ((\lambda c. \lambda f. f x c) c)) (A (R \bar{1} (\lambda c. \lambda f. f x c)) (K c))) \quad (R)
 \end{aligned}$$

$$\text{Since } \lambda c. H_0 A \Omega \bar{1} ((\lambda c. \lambda f. f x c) c) \rightarrow \lambda c. (\lambda_c w. \lambda c. w c) ((\lambda c. \lambda f. f x c) c) \quad (H), (A), (K)$$

$$\rightarrow \lambda c. x c =_{\eta} x$$

$$\text{and } A (R \bar{1} (\lambda c. \lambda f. f x c)) (K c) \xrightarrow[w]{} c \quad (A), (K)$$

$$\text{then } \Delta = \lambda_c x. \lambda c. x (\lambda f. f x c)$$

□

All reductions taking place in the head reduction of the source expression are performed on the transformed expression by weak head reduction. Here the resulting expression is interconvertible with the principal hnf in cps form. Note that the reconstruction process is not completed by weak head reduction. In a sense, the reconstruction process is lazy; it can take place (by wh-reduction) only when the resulting expression is applied. Only the required subexpressions will be reconstituted.

The following property states that for any closed expression E the weak head reduction of $\mathcal{N}(E)$ $A \Omega \bar{0}$ simulates the head reduction of E . If E has a hnf H then the wh-reduction of $\mathcal{N}(E) A \Omega \bar{0}$ yields an expression equal to $\mathcal{N}(H) A \Omega \bar{0}$ (after administrative reductions).

Property 2 *Let E be a closed expression, if $E \xrightarrow{*} H$ then there exists an expression X such that $\mathcal{N}(E) A \Omega \bar{0} \xrightarrow[w]{} X \xrightarrow{\ddagger} \mathcal{N}(H) A \Omega \bar{0}$ and if H is a hnf then X is a whnf. Furthermore E does not have a hnf iff $\mathcal{N}(E) A \Omega \bar{0}$ does not have a whnf.*

Proof. [Sketch] We first show that the property holds for one reduction step. Two lemmas are needed: “ $\mathcal{N}(E)[\mathcal{N}(F)/x] \equiv \mathcal{N}(E[F/x])$ ” which is shown in [16] and “if $x \neq y$ and x does not occur free in G then $E[F/x][G/y] \equiv E[G/y][F(G/y)/x]$ ” which is shown in [3] (2.1.16 pp. 27). The property is then shown by induction on the number of reduction steps. Concerning the second part of the property: if an expression E_0 does not have a hnf there is an infinite reduction sequence $E_0 \xrightarrow{H} E_1 \xrightarrow{H} \dots$. It is clear from the preceding proof that the corresponding weak head reduction on $\mathcal{N}(E_0) A \Omega \bar{0}$ will also be infinite, so this expression does not have a whnf. If E has a hnf H then $E_0 \xrightarrow{H} H$ so there is a X such that $\mathcal{N}(E_0) A \Omega \bar{0} \xrightarrow{W} X \xrightarrow{A} \mathcal{N}(H) A \Omega \bar{0}$. H being of the form $\lambda x_1 \dots x_n. x_i E_1 \dots E_p$, after administrative reductions, $\mathcal{N}(H) A \Omega \bar{0}$ is of the form $H_i C \Omega \bar{n}$ and the reduction rule of H_i yields a whnf. \square

In general $\mathcal{N}(H) A \Omega \bar{0} \leftrightarrow \mathcal{N}(H)$ does not hold, namely the result is not always interconvertible with the hnf in cps. This is usual with this kind of transformation ; the result is in compiled form and is convertible to its source version only under certain conditions. Still, $\mathcal{N}(H) A \Omega \bar{0}$ and $\mathcal{N}(H)$ have a strong relationship. Let $H \equiv \lambda x_1 \dots x_n. x_i E_1 \dots E_p$ then

$$\mathcal{N}(H) = \lambda_c \vec{x}_n . \lambda c. x_i (\mathcal{N}(E_p) c)$$

$$\text{and } \mathcal{N}(H) A \Omega \bar{0} = \lambda_c \vec{x}_n . \lambda c. x_i (\vec{X}_p E) \text{ with } X_i \equiv \lambda c. \mathcal{N}(\lambda \vec{x}_n . E_i) A \Omega \bar{0} (\vec{x}_n c).$$

So, the head variable is the same and if the sub-expressions $\mathcal{N}(E_i)$ and X_i have a hnf they will also have the same head variable. Likewise, if a sub-expression $\mathcal{N}(E_i)$ does not have a hnf then the corresponding expression $\mathcal{N}(\lambda \vec{x}_n . E_i) A \Omega \bar{0} (\vec{x}_n c)$ does not have a whnf ; they can then be considered equivalent. However we do not have a plain equivalence since there are expressions whose sub-expressions all have a hnf but have no nf themselves ; for example $(\lambda xy.y(xx))(\lambda xy.y(xx)) \rightarrow \dots \rightarrow (\lambda y.y(\lambda y.y \dots (\lambda xy.y(xx))(\lambda xy.y(xx))))$. For such expressions $\mathcal{N}(H) A \Omega \bar{0}$ and $\mathcal{N}(H)$ are not interconvertible; the H_i 's substituted for the leading variables may never be completely removed. However, for expressions with a normal form the following result holds.

Property 3 *Let E be a closed expression with a normal form then $\mathcal{N}(E) \leftrightarrow \mathcal{N}(E) A \Omega \bar{0}$*

Proof. [Sketch] If $E \rightarrow F$ then $\mathcal{N}(E) \xrightarrow{\Delta} \mathcal{N}(F)$ and then obviously $\mathcal{N}(E) A \Omega \bar{0} \rightarrow \mathcal{N}(F) A \Omega \bar{0}$ (just pick up the same redex). So if E has a normal form S then $E \xrightarrow{\Delta} S$ and $\mathcal{N}(E) A \Omega \bar{0} \xrightarrow{\Delta} \mathcal{N}(S) A \Omega \bar{0}$. We just have to show that for any normal form S , $\mathcal{N}(S) \leftrightarrow \mathcal{N}(S) A \Omega \bar{0}$ which is proved by induction on the structure of nfs. \square

Here, we propose one possible definition of combinators \bar{n} , Ω , H_i , R in terms of pure λ -expressions. We do not claim it is the best one ; we just want to show that such combinators can indeed be implemented in the same language. Simpler definitions could be conceived in a less rudimentary language (e.g. λ -calculus extended with constants).

We represent \bar{n} by Church integers, i.e. $\bar{0} = \lambda fx.x$ and $\bar{n} = \lambda fx.f^n x$. The successor function S^+ is defined by $S^+ = \lambda xyz.y(xy z)$.

$I = \lambda x.x$, $K = \lambda xy.x$, $A = \lambda xy.yx$ and $Y = (\lambda xy.y(xx))(\lambda xy.y(xx))$ (Turing's fixed point combinator)

$$\Omega = Y(\lambda wen. e(H n) A w(S^+ n))$$

The family H_i is represented by $H \bar{i}$ with

$$H = \lambda ieon. n L(\lambda ac.a I(W i)(e(R n a)(K c))) I$$

$$\text{where } W = \lambda i.i(\lambda xyz.zx) K$$

$$L = \lambda ab.\lambda_c x.a(\lambda c.b(\lambda f.f x c))$$

$$\text{and } R = Y(\lambda ruvwxy.\lambda f.f(\lambda c.w A \Omega u(v c))(x(r u v) y))$$

We can easily check that these definitions imply the reduction rules previously assumed, for example $\Omega E n \xrightarrow{w} E H_n A \Omega n+1$ or $R E F G H I \xrightarrow{w} \lambda f.f (\lambda c.G A \Omega E (F c)) (H (R E F) I)$.

4 Strong Reduction

Full normal forms are evaluated by first reducing expressions to hnf and then reducing the arguments of the head variable. We follow the same idea as for head reduction. Instead of instantiating variables by combinators H_i we use the family S_i which will carry out the evaluation before reconstructing. The recursive continuation Ω is the same as before except that it applies the λ -abstraction to S_i instead of H_i .

If E is a closed expression, its transformed form will be $\mathcal{N}(E) A \Omega \bar{0}$ with

$$\Omega M \bar{n} \xrightarrow{w} M S_n A \Omega \bar{n+1} \quad (\Omega)$$

and $S_i M N \bar{n} \xrightarrow{w} M E_n B H_i N \bar{n}$ (S)

where $E_i M N P \xrightarrow{w} N E_i P (M A \Omega \bar{i} C)$ (E)

$$B M N \xrightarrow{w} N A \quad (B)$$

$$C M N P \xrightarrow{w} P (\lambda f.f (\lambda c. c M) N) \quad (C)$$

When the hnf is reached the head variable previously instantiated by S_i is called. It triggers the evaluation of its arguments via E_i and insert H_i as last continuation. E_i applies the arguments to $A \Omega \bar{i}$ which will be evaluated in a right to left order and inserts the continuation C needed to put back the evaluated arguments X_1, \dots, X_n in cps form (i.e. $\lambda f.f X_1 (\dots (\lambda f.f X_n E) \dots)$). The role of H_i 's is still to reconstruct the expression. Combinator H_i keeps the same definition except for R which have now the simplified reduction rule

$$R E F G H I = \lambda f.f (\lambda c.G (F c)) (H (R E F) I) \quad (R)$$

When R is applied, the arguments are already evaluated and reconstructed so there is no need to apply them to $A \Omega \bar{i}$ as before.

Example: Let $E = \lambda x.(\lambda w.\lambda y.w y ((\lambda v.v) x)) (\lambda z.z) x$

Its strong reduction is

$$\begin{aligned} E &\xrightarrow{s} \lambda x.(\lambda y.(\lambda z.z) y ((\lambda v.v) x)) x \\ &\xrightarrow{s} \lambda x.(\lambda z.z) x ((\lambda v.v) x) \\ &\xrightarrow{s} \lambda x.x ((\lambda v.v) x) \\ &\xrightarrow{s} \lambda x.x x \end{aligned}$$

After cps conversion and simplification the expression becomes

$$\mathcal{N}(E) = \lambda_c x.(\lambda_w.\lambda_c y.\lambda_c w (\lambda f.f y (\lambda f.f ((\lambda v.v) x) c)) (\lambda_c z.z) (\lambda f.f x c))$$

The weak head reduction of the cps expression is (reductions corresponding to strong reductions of the source expression are marked by \dagger)

$$\begin{aligned}
& \mathcal{N}(E) A \Omega \bar{0} \xrightarrow{\frac{3}{W}} (\lambda x. \lambda c. (\lambda w. \lambda_c y. \lambda c. w (\lambda f. f y (\lambda f. f ((\lambda v. v) x) c)) (\lambda_c z. z) (\lambda f. f x c)) S_0 A \Omega \bar{1} \\
& \xrightarrow{\frac{2}{W}} (\lambda w. \lambda_c y. \lambda c. w (\lambda f. f y (\lambda f. f ((\lambda v. v) S_0) c)) (\lambda_c z. z) (\lambda f. f S_0 A) \Omega \bar{1} \\
& \xrightarrow{\frac{1}{W}} (\lambda_c y. \lambda c. (\lambda_c z. z) (\lambda f. f y (\lambda f. f ((\lambda v. v) S_0) c)) (\lambda f. f S_0 A) \Omega \bar{1} \quad + \\
& \xrightarrow{\frac{2}{W}} (\lambda y. \lambda c. (\lambda_c z. z) (\lambda f. f y (\lambda f. f ((\lambda v. v) S_0) c)) S_0 A \Omega \bar{1} \\
& \xrightarrow{\frac{1}{W}} (\lambda c. (\lambda_c z. z) (\lambda f. f S_0 (\lambda f. f ((\lambda v. v) S_0) c)) A \Omega \bar{1} \quad + \\
& \xrightarrow{\frac{3}{W}} (\lambda z. z) S_0 (\lambda f. f ((\lambda v. v) S_0) A) \Omega \bar{1} \\
& \xrightarrow{\frac{1}{W}} S_0 (\lambda f. f ((\lambda v. v) S_0) A) \Omega \bar{1} \quad \text{the hnf is reached, the reduction rule of } S_0 \text{ is used.} + \\
& \xrightarrow{\frac{1}{W}} (\lambda f. f ((\lambda v. v) S_0) A) E_1 B H_0 \Omega \bar{1} \\
& \xrightarrow{\frac{1}{W}} E_1 ((\lambda v. v) S_0) A B H_0 \Omega \bar{1} \\
& \xrightarrow{\frac{3}{W}} ((\lambda v. v) S_0 A \Omega \bar{1} C) A H_0 \Omega \bar{1} \\
& \xrightarrow{\frac{1}{W}} S_0 A \Omega \bar{1} C A H_0 \Omega \bar{1} \quad \text{the nf is reached ; the reconstruction begins.} + \\
& \xrightarrow{\frac{3}{W}} H_0 A \Omega \bar{1} C A H_0 \Omega \bar{1} \\
& \xrightarrow{\frac{1}{W}} (\lambda_c x. \lambda c. x (A (R \bar{1} (\lambda c. \lambda f. f x c)) (K c))) C A H_0 \Omega \bar{1} \\
& \xrightarrow{\frac{1}{W}} C (\lambda x. \lambda c. x (A (R \bar{1} (\lambda c. \lambda f. f x c)) (K c))) A H_0 \Omega \bar{1} \\
& \xrightarrow{\frac{1}{W}} H_0 X \Omega \bar{1} \quad \text{with } X \equiv \lambda f. f (\lambda c. c (\lambda x. \lambda c. x (A (R \bar{1} (\lambda c. \lambda f. f x c)) (K c)))) A \\
& \rightarrow \lambda_c x. \lambda c. x (X (R \bar{1} (\lambda c. \lambda f. f x c)) (K c))
\end{aligned}$$

The wh-reduction is completed. Now, we show that the result is equivalent to the normal form in cps form.

$$X \xrightarrow{*} \lambda f. f (\lambda_c x. x) A \quad (A), (K)$$

$$\text{and } X (R \bar{1} (\lambda c. \lambda f. f x c)) (K c) \xrightarrow{*} R \bar{1} (\lambda c. \lambda f. f x c) (\lambda_c x. x) A (K c)$$

$$\xrightarrow{*} \lambda f. f (\lambda c. (\lambda_c x. x) ((\lambda c. \lambda f. f x c) c)) (A (R \bar{1} (\lambda c. \lambda f. f x c)) (K c)) \quad (R)$$

$$\xrightarrow{*} \lambda f. f x c \quad \text{since} \quad A (R \bar{1} (\lambda c. \lambda f. f x c)) (K c) \xrightarrow{*} c \quad (A), (K)$$

$$\text{and} \quad \lambda c. (\lambda_c x. x) ((\lambda c. \lambda f. f x c) c) \rightarrow \lambda c. x c \rightarrow_{\eta} x$$

So $\lambda_c x. \lambda c. x (X (R \bar{1} (\lambda c. \lambda f. f x c)) (K c)) \xrightarrow{*} \lambda_c x. \lambda c. x (\lambda f. f x c)$ which is the normal form in cps form. \square

All the reductions taking place during the strong reduction of the source expression are carried out by wh-reduction of the transformed expression. We do not really get the full normal form since the reconstruction can not be achieved completely by weak head reduction. As before the reconstruction is lazy. However the result is convertible to the normal form in cps and the complexity of this last step is bounded by the size of the normal form. If we were just interested in normal forms as a syntactic result, H_i 's could be replaced by functions printing the nf instead of building a suspension representing it. In this case, the evaluation would be completely carried out by wh-reduction.

We have the analogues of Property 2 and Property 3. The following property states that for any closed expression E the weak head reduction of $\mathcal{N}(E) A \Omega \bar{0}$ simulates the strong reduction of E .

Property 4 Let E be a closed expression, if $E \xrightarrow{*} S$ then there exists an expression X such that $\mathcal{N}(E) A \Omega \bar{0} \xrightarrow{*_{\bar{W}}} X \xrightarrow{\Delta} \mathcal{N}(S) A \Omega \bar{0}$ and if S is a nf then X is a whnf. Furthermore, E does not have a nf iff $\mathcal{N}(E) A \Omega \bar{0}$ does not have a whnf.

The result of the evaluation of $\mathcal{N}(E) A \Omega \bar{0}$ is interconvertible with the nf in cps.

Property 5 If a closed expression E has a normal form then $\mathcal{N}(E) \leftrightarrow \mathcal{N}(E) A \Omega \bar{0}$

Their proofs are similar to those of Property 2 and Property 3.

5 Head Reduction of Typed λ -Expressions

In the previous sections we needed to count the number of leading λ 's during the evaluation. Using some form of typing it is possible to know the functionality of the expression prior to evaluation and thus get rid of this counter. We consider only head reduction ; typing does not seem to simplify the compilation of strong reduction.

Simply typed λ -calculus would suit our purposes but would harshly restrict the class of expressions. More flexible typing systems are sufficient. One candidate is reflexive reducing typing [2] which has already been used in [9] to determine the functionality of expressions. It is shown in [2] that we can restrict a language to reflexive reducing types without weakening its expressive power. Reflexive reducing types are defined by (possibly recursive) equations of the form $\sigma = \sigma_1 \rightarrow \dots \rightarrow \sigma_n \rightarrow \alpha$, $\sigma_1, \dots, \sigma_n$ being themselves reflexive reducing types and α being a base type (not a reflexive type). This enables us to type recursive functions but not for example $(\lambda xy.xx)(\lambda xy.xx)$ (this expression has the reflexive type $p \rightarrow \alpha \rightarrow \sigma$ with $\sigma = \alpha \rightarrow \sigma$ which is not reducing). We do not dwell here on the details of this typing system. The important point for us is that if a closed expression with type $\sigma = \sigma_1 \rightarrow \dots \rightarrow \sigma_n \rightarrow \alpha$ has a hnf then it is of the form $\lambda x_1. \dots \lambda x_n. x_i E_1 \dots E_p$.

If the expression to reduce to hnf has functionality n then the transformed expression is

$$\mathcal{N}(E) (\lambda f.f X_n^1 \dots (\lambda f.f X_n^n L_n) \dots) \text{ and we note } \mathcal{N}(E) (\vec{X}_n \vec{L}_n).$$

That is, we apply the expression to n arguments in order to remove the n leading abstractions. Combinators X_n^i play the same role as the combinators H_i introduced in section 3. They will be substituted for variables and used to start the reconstruction process.

$$X_n^i E = \lambda_c \vec{X}_n . \lambda c. x_i (E (R_n (\lambda c. \vec{X}_n c)) (K c)) \quad (X)$$

Combinator R_n used in the definition of X_n^i plays the same role as R in the definition of H_i .

$$R_n E F G H = \lambda f.f (\lambda c.F L_n (E c)) (G (R_n E) H) \quad (R)$$

In the preceding sections, the reconstruction of subexpressions was based on the same technique as the reduction of the global expression: each subexpression was applied to continuation Ω and was rebuild after being reduced to hnf. Here, there is no type information available on the subexpressions and we cannot use the same method as for the global expression. In particular, a subexpression $(\lambda c z. E)$ can not be reduced since we do not know its functionality. However, it may contain occurrences of combinators X_n^i which are to be removed. This case is treated using combinators L_n and Z_n which carry on the reconstruction inside the λ -abstraction.

$$L_n E = \lambda_c \vec{X}_n . \lambda c. z. \lambda c. E (Z_n z) L_n (\vec{X}_n c) \quad (L)$$

$$Z_n E F = \lambda_c \vec{X}_n . \lambda c. E (F (R_n (\lambda c. \vec{X}_n c)) (K c)) \quad (Z)$$

For example, if the hnf is of the form $\lambda x_1. \dots \lambda x_n. x_i \dots (\lambda z. E) \dots$ then R_n applies each subexpression to L_n and $(\vec{x}_n \ c)$ and we will get for the λ -abstraction $(\lambda z. E)$

$$\begin{aligned} \lambda c. (\lambda_c z. E) L_n (\vec{x}_n \ c) &\rightarrow \lambda c. L_n (\lambda z. E) (\vec{x}_n \ c) \\ &\rightarrow \lambda c. (\lambda_c \vec{x}_n . \lambda_c z. \lambda c. (\lambda z. E) (Z_n z) L_n (\vec{x}_n \ c)) (\vec{x}_n \ c) \\ &\rightarrow \lambda_c z. \lambda c. E[Z_n z/z] L_n (\vec{x}_n \ c) \end{aligned}$$

The list of variables has been pushed inside the λ -abstraction and the reconstruction can continue. Variable z is replaced by $(Z_n z)$ so that when it is applied to the list $(\vec{x}_n \ c)$ it returns z . Combinators R_n , L_n , X_n^i , Z_n act very much like combinators used in abstraction algorithms. X_n^i is a selector (it selects the i th variable), Z_n is (like K) a destructor (it ignores the list and returns its first argument), R_n and L_n distribute the list of variables $(\lambda c. x_n \ c)$ throughout the expression.

The head normal form (if any) of E will be of the form $(\lambda x_1 \dots \lambda x_n. x_i E_1 \dots E_p)$ so $\mathcal{N}(E) (\vec{X}_n \ L_n)$ will be reduced (by weak head reduction) to $X_n^i (\vec{E}_p \ L_n)$ and then, according to the definition of combinators X_n^i , to $\lambda_c x_n . \lambda c. x_i ((\vec{E}_p \ L_n) (R_n (\lambda c. x_n \ c)) (K c))$. As before the continuation $(K c)$ removes reconstructing expressions and returns the final continuation.

The following property states that for any closed expression E of type $\sigma_1 \rightarrow \dots \rightarrow \sigma_n \rightarrow \alpha$ the weak head reduction of $\mathcal{N}(E) (\vec{X}_n \ L_n)$ simulates the head reduction of E .

Property 6 *Let E be a closed expression of functionality n , if $E \xrightarrow{*} H$ then there exists an expression X such that $\mathcal{N}(E) (\vec{X}_n \ L_n) \xrightarrow{*} X \xleftarrow{W} \mathcal{N}(H) (\vec{X}_n \ L_n)$ and if H is a hnf then X is a whnf. Furthermore, E does not have a hnf iff $\mathcal{N}(E) (\vec{X}_n \ L_n)$ does not have a whnf.*

If E has a normal form the result of the evaluation of $\mathcal{N}(E) (\vec{X}_n \ L_n)$ is interconvertible with the principal hnf in cps form.

Property 7 *If a closed expression E of functionality n has a normal form then $\mathcal{N}(E) \leftrightarrow \mathcal{N}(E) (\vec{X}_n \ L_n)$*

Their proofs are similar to those of Property 2 and Property 3.

6 Applications

Among practical applications of head reduction listed in the introduction, one is to compile more efficient evaluation strategies. We describe better this question in the next section and suggest in section 6.2 how our approach can be used to compile such strategies. Implementation issues are discussed in 6.3.

6.1 Spine Strategies

Even when evaluating weak-head-normal forms it is sometimes better to reduce sub-terms in head normal forms. For example, in lazy graph reduction, the implementation of β -reduction $(\lambda x. E) F \rightarrow_{\beta} E[F/x]$ implies making a copy of the body E before the substitution. It is well known that this may lose sharing and work may be duplicated [18]. Program transformations, such as fully lazy lambda-lifting [10], aim at maximizing sharing but duplication of work can still occur. Another approach used to avoid recomputation is to consider alternative evaluation strategies. If the expression to reduce is $(\lambda x. E) F$ we know that the whnf of the body E will be needed and so it is safe to reduce E prior to the β -reduction. This computation rule belongs to the so-called spine-strategies [4]. It never takes more reductions than normal order and may prevent duplication of work.

A revealing example, taken from [8], is the reduction of $A_n I$ where the family of λ -expressions A_i is defined by $A_0 = \lambda x. x I$ and $A_n = \lambda h. (\lambda w. w h (w w)) A_{n-1}$. The expression $A_n I$ is reduced using the call-by-name weak head graph reduction as follows:

$$\begin{aligned}
 A_n I &= (\lambda h.(\lambda w.w\ h\ (w\ w))\ A_{n-1})\ I \\
 &\rightarrow (\lambda w.w\ I\ (w\ w))\ A_{n-1} \\
 &\rightarrow A_{n-1}\ I\ (\bullet\ \bullet) \equiv (\lambda h.(\lambda w.w\ h\ (w\ w))\ A_{n-2})\ I\ (\bullet\ \bullet) \quad (\bullet \text{ representing the sharing of } A_{n-1}) \\
 &\rightarrow (\lambda w.w\ I\ (w\ w))\ A_{n-2}\ (A_{n-1}\ \bullet)
 \end{aligned}$$

The sharing is lost and the redexes inside A_{n-1} are duplicated. The complexity of the evaluation is $O(2^n)$. On the other hand, by reducing λ -abstractions to hnf before β -reductions the evaluation sequence becomes

$$\begin{aligned}
 A_n I &= (\lambda h.(\lambda w.w\ h\ (w\ w))\ A_{n-1})\ I \\
 &\rightarrow (\lambda h.A_{n-1}\ h\ (\bullet\ \bullet))\ I \\
 &\xrightarrow{4(n-1)} (\lambda h.A_0\ h\ (\bullet\ \bullet))\ I \quad A_{n-1} \text{ reduces to } A_0 \text{ in } 4(n-1) \text{ steps} \\
 &\rightarrow (\lambda h.\ I\ (A_0\ \bullet))\ I \rightarrow (\lambda h.\ A_0\ \bullet)\ I \rightarrow (\lambda h.\ I)\ I \rightarrow I
 \end{aligned}$$

and the A_i 's remain shared until they are reduced to their hnf A_0 . The complexity of the evaluation drops from exponential to linear.

Of course this strategy alone is not optimal (optimal reduction of λ -expressions is more complex [12][13]) and work can still be duplicated. But in [17] Staples proposes a similar evaluation strategy with the additional rule that substitutions are not carried out inside redexes (they are suspended until the redex is needed and reduced to hnf). This reduction has been shown to be optimal for a λ -calculus with explicit substitutions.

6.2 Sharing Hnf's

We saw that evaluating the λ -abstraction to hnf before the β -reduction can save work by sharing hnf's instead of whnf's. Following this idea, maximal sharing is obtained by reducing every closure to hnf instead of whnf. The straightforward idea of applying closures to $A\ \Omega\ \bar{0}$ does not work. Our previous results were relying on the fact that the expression to be reduced was closed. Here, even if the global expression is closed, we may have to reduce to hnf sub-expressions containing free variables. For example, if $(\lambda x.\ I\ (\lambda y.\ I\ x))$ is cps converted and the two closures $(\lambda_c x. \dots)$ and $(\lambda_c y. \dots)$ are applied to $A\ \Omega\ \bar{0}$ then during the reduction of $(\lambda_c x. \dots)$ we will have to reduce to hnf $(\lambda_c y. \dots)\ A\ \Omega\ \bar{0}$. But x is already instantiated by H_0 and we get $(\lambda_c y. H_0)\ A\ \Omega\ \bar{0} \rightarrow H_0\ A\ \Omega\ \bar{1} \rightarrow (\lambda_c y. y)$ which is false. The cps hnf of $(\lambda y.\ I\ x)$ should have been $(\lambda_c y. H_0)$ and the enclosing evaluation of $(\lambda_c x. \dots)$ could continue. The problem comes from free variables already instantiated by combinators when a new head reduction begins.

One solution to the free variable problem is to use a second index as in [5]. In our framework, this technique is expressed by changing the rule of cps conversion for applications

$$\mathcal{N}^*(E\ F) = \lambda cwn.\mathcal{N}^*(E)\ (\lambda f.f\ (\mathcal{N}^*(F)\ A\ \Omega\ n\ n)\ c)\ w\ n$$

Each closure is applied to two indexes, initially the current binder length. The first one will play the same role as before and will increase at each leading lambda encountered during the reduction of the closure. The second index, say k , remains fixed for each closure and is used to determine if a combinator H_i corresponds to a free variable ($i < k$) or a bound variable ($i \geq k$) in that context. The definitions of combinators H_i and R become

$$H_i M N \bar{n} \bar{k} = \lambda_c x_{n-k} \cdot \lambda_{cwn} x_{i-k+1} (M (R \bar{n} \bar{k} n (\lambda_c x_{n-k} c)) (K c)) w n , \text{ if } i \geq k \quad (H1)$$

$$= \lambda_c x_{n-k} \cdot \lambda_{cwn} H_i (M (R \bar{n} \bar{k} n (\lambda_c x_{n-k} c)) (K c)) w n , \text{ if } i < k \quad (H2)$$

$$R C D E F G H I \xrightarrow{w} \lambda f.f (G A \Omega C D (F A)) \Omega E E (H (R C D E F) I)$$

Now, the reduction of the (cps form of the) closure ($\lambda x. I (\lambda y. I x)$) becomes

$$(\lambda_c x. \lambda_{cwn} I ((\lambda_c y. I x) A \Omega n n) c) w n A \Omega \bar{0} \bar{0}$$

$$\xrightarrow{*} (\lambda_{cwn} I ((\lambda_c y. I H_0) A \Omega n n) c w n) A \Omega \bar{1} \bar{0}$$

$$\xrightarrow{*} (\lambda_c y. I H_0) A \Omega \bar{1} \bar{1} A \Omega \bar{1} \bar{0}$$

$$\xrightarrow{*} H_0 A \Omega \bar{2} \bar{1} A \Omega \bar{1} \bar{0}$$

H_0 corresponds to a free variable in $(\lambda_c y. \dots)$

$$\xrightarrow{*} (\lambda_c y. H_0) A \Omega \bar{1} \bar{0}$$

(H2)

$$\xrightarrow{*} H_0 A \Omega \bar{2} \bar{0} \rightarrow \lambda_c x. \lambda_{cwn} I ((\lambda_c y. I x) A \Omega n n) c \quad H_0 \text{ corresponds to a bound variable in } (\lambda_c x. \dots) \quad (H1)$$

Several optimizations can be designed to avoid producing useless closures. An important one (that we used in the example above) is $\mathcal{N}^*(E x) = \lambda c. \mathcal{N}^*(E) (\lambda f. f x c)$. This rule holds because a variable is always instantiated either by a combinator H_i or by a closure $\mathcal{N}^*(F) A \Omega m m$. It can be shown that $H_i A \Omega n n = H_i$ (if $i < n$) and $(\mathcal{N}^*(F) A \Omega m m) A \Omega n n = \mathcal{N}^*(F) A \Omega m m$ if $n \geq m$.

Sharing as much hnf's as possible is likely to be quite costly in practice. In [5] Crégut gives a function for which the reduction takes n^2 steps when sharing hnf's whereas it takes only n steps using standard wh-reduction. It is also shown that this is the worst case. Some less extreme strategies may be envisaged. For example, the imbrication of several head reductions could be forbidden (i.e. closures would be reduced to hnf by the top level reduction but only to whnf during the reduction of another closure). This would simplify the reduction but the price is a potential loss of sharing.

6.3 Implementation issues

The most obvious way to implement our approach is to transform expressions as previously described and give the result to a compiler. The combinators A, Ω, \dots are compiled like other functions and the reconstruction is naturally implemented by closure building. However, with compilers which already integrate a cps conversion, a more efficient way would be to directly use the cps phase. This is less trivial since the following steps expect only cps expressions and we have to introduce special combinators which are not in cps. One solution is to implement those combinators by hand and the compiler can use them as primitive functions. We plan such an integration in our cps-based compiler. Further work is still needed on different extensions:

- So far we have only considered call by name. As cps conversion can be used to compile different computation rules (call-by-value, call-by-name with strictness annotations, ...) it is likely that our method could be extended to treat those strategies as well.
- This method should be extended to a λ -calculus with constants and primitive operators.

If we just aim at reducing a program to hnf/nf and print the result then our approach will be very efficient. The whole evaluation is a weak reduction which can be completely compiled. The only slight overhead will be a few more reductions for each leading lambda and printing the result which should be proportional to the size of the expression.

The costly part of head/strong reduction is the reconstruction of expressions which happens when we actually use (i.e. apply) the hnf/nf. In particular, reconstructing uses a lot of memory space. In order to implement efficient evaluation strategies as described previously, it would be useful to develop the following points:

- Several analyses can detect expressions for which wh-reduction is better and should be implemented as well. For example, one policy could be that a closure will be reduced to hnf only if it is shared (using a sharing analysis), complex enough (using a complexity analysis) and of course not already in hnf.
- Computation can still be duplicated by performing substitutions inside redexes. It would be interesting to extend our work to compile Staples' method [17] which avoids this loss of sharing.

We did a few experiments using the trivial way (i.e. transforming source expressions before giving them to our compiler). We transformed the family of expressions A_n defined in section 6.1 into supercombinators and into cps form. The evaluation of $A_{15} I$ takes around 1s using standard reduction and around 1ms when each supercombinator is applied to $A \Omega 0$. This result is not surprising since the theoretical complexity is exponential in one case and linear in the other. More interestingly, we redefined the family A_n by $A_0 = \lambda x.x I$ and $A_n = \lambda h.(\lambda w.w h (A_0 w)) A_{n-1}$. Here, the wh-reduction of $A_n I$ does not duplicate work (the second occurrence of w is not needed) and nothing is saved by using head reduction. We found that the head reduction of supercombinators made the evaluation 3 to 4 times slower than the standard wh-reduction. This example indicates the cost of reconstructing expressions. This cost is acceptable when the final goal is to implement symbolic evaluation. When the goal is to evaluate whnf's more efficiently by sharing hnf's then such examples should be avoided using analyses or (maybe more pragmatically) using user's annotations.

7 Conclusion

Implementation of head and strong reduction has also been studied by Crégut [5] and Nadathur and Wilson [14]. Crégut's abstract machine is based on De Bruijn's notation. Two versions have been developed. The first one evaluates the head or full normal form of the global expression. The second one implements a spine strategy and shares head normal forms. Terms are extended with formal variables and the machine state includes two indexes. One plays the role of our binder level as in section 3 and 4, the other one is needed (only in the second version of the machine) to deal with the problem of free variables in subexpressions as exposed in section 6.2. The algorithm presented in [14] was motivated by the implementation of λ Prolog [15]. It evaluates terms to hnf and, expressed as an abstract machine, this technique resembles Crégut's. It is also based on De Bruijn notation and the machine state includes two indexes.

We described in this paper how to use cps conversion to compile head and strong reduction. The hnf's or nf's of cps-expressions are evaluated by weak head reduction and at each step the unique (weak) redex is the leftmost application. The technique does not require to modify the standard cps cbn conversion. The cps expression is just applied to a special continuation and an index to keep track of the binder length. We presented a way to get rid of this index and suggested applications for our technique. Cps conversion was important to this work in several respects: special continuations could be used to suppress the leading λ 's and the regular form of cps expressions helped the reconstruction.

Compared to [5] and [14] the main difference is that we proceed by program transformations and stay within the functional framework. Used as a preliminary step our technique allows a standard compiler to evaluate under λ 's. Thus we can take advantage of all the classical compiling tools like analyses, transformations or simplifications. As already emphasized in [7], another advantage of this approach is that we do not have to introduce an abstract machine which makes correctness proofs simpler. Furthermore, optimizations of this compilation step can be easily expressed and justified in the functional framework.

Apart from the practical issues discussed in section 6.3, several others research directions like the application of this approach to partial evaluation or to the compilation of λ -prolog should be explored.

Acknowledgments. Thanks are due to Daniel Le Métayer for his comments on an earlier version of this paper.

References

- [1] A. W. Appel. *Compiling with Continuations*. Cambridge University Press. 1992.
- [2] E. Astesiano and G. Costa. Languages with reducing reflexive types. In *7th Coll. on Automata, Languages and Programming*, LNCS Vol. 85, pp. 38-50, 1980.
- [3] H.P. Barendregt. *The Lambda Calculus. Its Syntax and Semantics*. North-Holland, 1981.
- [4] H.P. Barendregt, J.R. Kennaway, J.W. Klop and M.R. Sleep. Needed reduction and spine strategies for the lambda calculus. *Information and Computation*, Vol. 75, pp. 191-231, 1987.
- [5] P. Crégut. An abstract machine for the normalization of λ -terms. In *Proc. of the ACM Conf. on Lisp and Functional Programming*, pp. 333-340, 1990.
- [6] M. J. Fischer. Lambda-calculus schemata. In *Proc. of the ACM Conf. on Proving Properties about Programs*, Sigplan Notices, Vol. 7(1), pp. 104-109, 1972. Revised version in *Lisp and Symb. Comp.*, Vol. 6, Nos. 3/4, 1993.
- [7] P. Fradet and D. Le Métayer. Compilation of functional languages by program transformation. *ACM Trans. on Prog. Lang. and Sys.*, 13(1), pp. 21-51, 1991.
- [8] G.S. Frandsen and C. Sturtivant. What is an efficient implementation of the λ -calculus? In *Proc. of the ACM Conf. on Functional Prog. Languages and Comp. Arch.*, LNCS Vol. 523, pp. 289-312, 1991.
- [9] M. Georgeff. Transformations and reduction strategies for typed lambda expressions. *ACM Trans. on Prog. Lang. and Sys.*, 6(4), pp. 603-631, 1984.
- [10] R.J.M. Hughes. Supercombinators, a new implementation method for applicative languages. In *Proc. of the ACM Conf. on Lisp and Functional Programming*, pp. 1-10, 1982.
- [11] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin and N. Adams. Orbit: An optimizing compiler for scheme. In *proc. of 1986 ACM SIGPLAN Symp. on Comp. Construction*, 219-233, 1986.
- [12] J. Lamping. An algorithm for lambda calculus optimal reductions. In *Proc. of the ACM Conf. on Princ. of Prog. Lang.*, pp. 16-30, 1990.
- [13] J.-J. Lévy. *Réductions correctes et optimales dans le lambda calcul*. Doctorat d'état, Paris VII, 1978.
- [14] G. Nadathur and D.S. Wilson. A representation of lambda terms suitable for operations on their intentions, In *Proc. of the ACM Conf. on Lisp and Functional Programming*, pp. 341-348, 1990.
- [15] G. Nadathur and D. Miller. An overview of λ Prolog. In *Proc. of the 5th Int. Conf. on Logic Prog.*, MIT Press, pp. 810-827, 1988.
- [16] G.D. Plotkin. Call-by-name, call-by-value and the λ -calculus. *Theoretical Computer Science*, pp. 125-159, 1975.
- [17] J. Staples. A graph-like lambda calculus for which leftmost-outermost reduction is optimal. In *Graph Grammars and their Application*, LNCS vol. 73, pp. 440-455, 1978.
- [18] C.P. Wadsworth. *Semantics and Pragmatics of the Lambda-Calculus*. PhD thesis, Oxford, 1971.

Suffix Trees in the Functional Programming Paradigm

Robert Giegerich *Stefan Kurtz*

Universität Bielefeld, Technische Fakultät, Postfach 100 131
D-33501 Bielefeld, Germany
e-mail: {robert,kurtz}@techfak.uni-bielefeld.de

Abstract. We explore the design space of implementing suffix tree algorithms in the functional paradigm. We review the linear time and space algorithms of McCreight and Ukkonen. Based on a new terminology of nested suffixes and nested prefixes, we give a simpler and more declarative explanation of these algorithms than was previously known. We design two “naive” versions of these algorithms which are not linear time, but use simpler data structures, and can be implemented in a purely functional style. Furthermore, we present a new, “lazy” suffix tree construction which is even simpler. We evaluate both imperative and functional implementations of these algorithms. Our results show that the naive algorithms perform very favourably, and in particular, the “lazy” construction compares very well to all the others.

1 Introduction

Suffix trees are the method of choice when a large sequence of symbols, henceforth called the “text”, is to be searched frequently for occurrences of short sequences, henceforth called “patterns”. Given that the text t is known and does not change (think of a famous novel or genetic data), while the patterns are not known in advance, one has to invest a certain effort to construct t ’s representation as a suffix tree. Given this suffix tree, all occurrences of a pattern p can be located in $\mathcal{O}(|p|)$ steps, independent of the length of t . This efficient access to all subwords of t have made suffix trees a ubiquitous data structure in a “myriad” of applications [1].

Since suffix tree construction is the price to be pre-paid, it is fortunate that suffix trees can be built in $\mathcal{O}(n)$ time and represented in $\mathcal{O}(n)$ space, where n is the length of t . Suffix tree construction algorithms have a long history, starting with [20]. The construction in that paper is given in a somewhat obscure terminology. Later authors [14, 12, 4] have developed more transparent constructions, sometimes tailored to specific additional requirements. The endpoint of the development is currently marked by [18] and [19], presenting a simpler construction in $\mathcal{O}(n)$ space and time, which additionally is *on-line*. It processes the string from left to right, and hence, in this sense it is incremental. What more can one ask for?

Our interest in suffix trees is motivated by our work on a flexible pattern-matching system for biosequence analysis [6]. Besides for locating subwords, suffix trees are useful for finding repetitions and palindromes, deriving q -gram profiles [17], and calculating the so-called matching statistics as a prerequisite for fast approximate matching [3].

Our system design follows a language-oriented rather than a tool-box approach. The user is provided a declarative language for describing pattern-matching problems. Sophisticated algorithms for “standard” problems are embedded in this language; suffix tree construction is one of these. Building on this machinery, complex matching problems are solved via backtracking. This approach leads to the following requirements: Our suffix tree construction should be embedded in a declarative language, polymorphic with respect to the underlying alphabet, and extensible with respect to application-specific annotation. The tree implementation should be as simple as possible, since this data structure will be visible to the user. A final desirable feature is incrementality, which has different and competing aspects. One aspect is incrementality with respect to the input text, i.e. *on-line* construction. The other aspect is that the suffix tree itself should be constructed incrementally as it is traversed, leaving incomplete those subtrees that are never actually needed, i.e. “lazy” construction.

Heretofore, suffix tree constructions have always been given in an imperative style. The best known algorithms heavily depend on local updates to the tree data structure, and hence violate the principle of statelessness. In this paper, we

- present a new, “lazy” construction for suffix trees, probably the simplest construction that has ever been given,
- review Ukkonen’s and McCreight’s $\mathcal{O}(n)$ -time suffix tree constructions¹ and derive simpler, but less efficient versions that can be implemented in a purely functional way,
- evaluate their efficiency in imperative and functional implementations,
- conclude with some observations about the benefits of studying the same algorithm in both the functional and the imperative paradigms.

For reasons of space, we omit all proofs and most of the formal development of the linear-time algorithms. All this can be found in [7].

2 Basic Notions

Let \mathcal{A} be a finite set, called alphabet. The elements of \mathcal{A} are called letters. ε denotes the empty string and \mathcal{A}^+ denotes the set of nonempty strings over \mathcal{A} . Let t be a string. A string v is called t -word, if and only if $t = uvw$ for some strings u and w . We call a t -word b branching, if and only if there are different letters x and y , such that bx and by are t -words. Let $t = uv$ for some strings u

¹ both are faster than the off-line constructions of [20] and [4], which are also $\mathcal{O}(n)$ (cf. [8]).

and v . Then u is a prefix of t and v is a suffix of t . Let $a \in \mathcal{A}$. A suffix of ta is called nested, if and only if it is a t -word. In other words, a nested suffix of ta has another occurrence as a t -word. (By convention ε is a nested suffix of ε .) Let s be a suffix of t . A prefix p of s is called nested, if and only if $p = \varepsilon$ or there is a suffix s' of t , such that $|s'| > |s|$ and p is a prefix of s' . In other words, a nested prefix is empty or has another occurrence as a prefix of a longer suffix of t .

3 The Suffix Tree Family

We give a rather liberal definition of suffix trees, and then three more restricted instances of it, called the naive suffix tree, the position suffix tree, and the linear suffix tree. We mention all three of them, since sometimes a construction with inferior theoretical worst case or average case space bounds may be superior in practice, due to its simpler construction or better speed of traversal.

An \mathcal{A}^+ -tree is a rooted tree with edge labels from \mathcal{A}^+ . For each $a \in \mathcal{A}$, a node has at most one a -edge $k \xrightarrow{au} k'$. By $path(k)$ we denote the concatenation of the edge labels on the path from the root to the node k . Due to the requirement of unique a -edges at each node, paths are also unique and we can denote k by \overline{w} , if and only if $path(k) = w$.² We say that a string u occurs in the tree, if and only if there is a node \overline{uv} , for some string v .

Definition 3.1 A suffix tree \mathcal{S}_t for a string t is an \mathcal{A}^+ -tree such that w occurs in \mathcal{S}_t , if and only if w is a t -word. \square

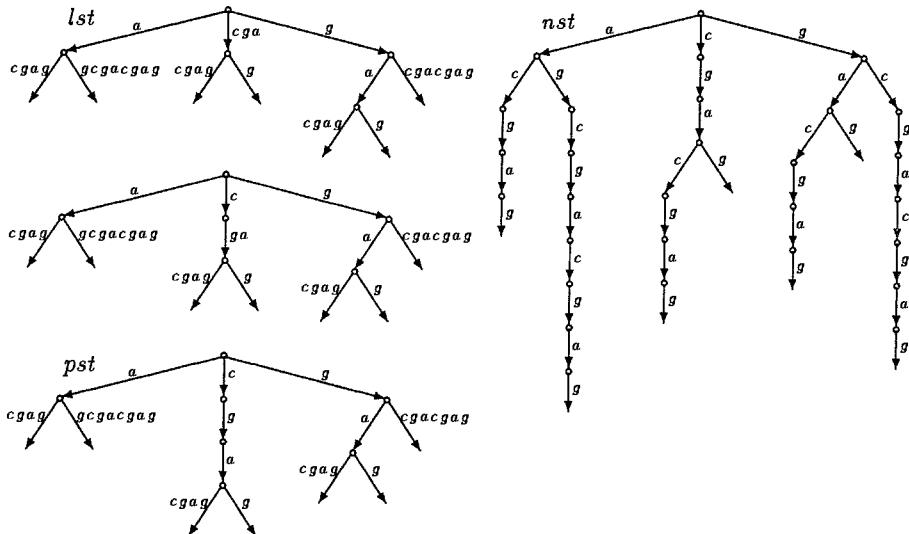
Definition 3.2 Let \mathcal{S}_t be a suffix tree for t , given by its edge set E .

1. If $|E|$ is maximal, \mathcal{S}_t is called naive suffix tree for t and denoted by $nst(t)$.
2. If for all $\overline{w} \xrightarrow{u} \overline{v} \in E$ either wa is not unique in t and $u = \varepsilon$ or wa is unique in t and \overline{v} is a leaf, then \mathcal{S}_t is called position suffix tree for t and denoted by $pst(t)$.
3. If $|E|$ is minimal, \mathcal{S}_t is called linear suffix tree for t and denoted by $lst(t)$.
 \square

$lst(t)$ has $\mathcal{O}(n)$ nodes, as all inner nodes are branching, and there are at most n leaves. The edge labels can be represented in constant space by a pair of indices into t . In practice, this is quite a delicate choice of representation in a virtual memory environment. Traversing the tree and reading the edge labels will create random-like accesses into the text, and can lead to severe paging problems.

² This is a most elegant, but also deceptive convention. It is easy to express relationships between tree nodes, e.g. \overline{cw} and \overline{w} , that are quite unrelated in the tree structure.

Fig. 1. Different suffix trees for the string $agcgacgag$.



4 Functional Suffix Tree Algorithms

In this section, we present functional suffix tree algorithms. The first one, called *lazyTree*, is new. The other two, called *naiveOnline* and *naiveInsertion* are simplified and less efficient versions of Ukkonen's and McCreight's algorithms. The simplification as well as the loss of efficiency result from the need to avoid local updates of the tree during its construction.

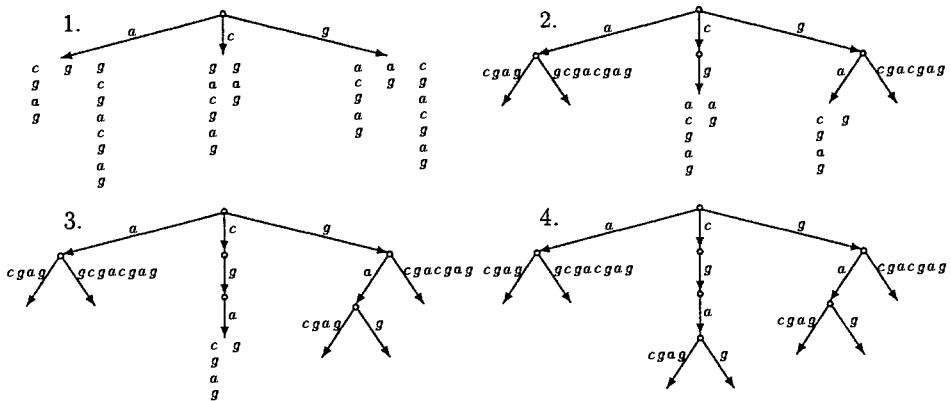
4.1 The Lazy Suffix Tree Construction

We call a suffix tree construction *lazy* when it constructs the suffix tree for the complete text from the *root* towards the leaves. This has the advantage that the construction may be interleaved with tree traversal – paths of the suffix tree need to be constructed (only) when being traversed for the first time. This kind of incrementality is achieved for free when implementing the lazy construction in a lazy language. It can be simulated in an eager language by explicit synchronization between construction and (all) traversal routines.

Aside from matters of efficiency, which are further discussed in section 4.4, our lazy construction is probably the simplest suffix tree construction that can be given.

We start with a graphical explanation: Write down the *root* with a sorted list l of all non-nested suffixes of t . Let $l_a = \{s \mid as \in l\}$, for each $a \in \mathcal{A}$. Then $pst(t)$ emerges by creating, for each nonempty l_a , an a -edge leading to the subtree recursively constructed for l_a .

Fig. 2. Phases of the lazy construction of the position suffix tree for *agcgacgag*.



Using a *sorted* list of suffixes only helps when doing this on paper – all the process needs is to group the suffixes according to their first character, and then choose a common prefix of each group for the edge label. This construction is reflected literally in the functional implementation shown below.³

```
sTree * ::= Leaf | Branch [[*,sTree *]]
edgeFunction * == [[*]]->([*],[[*]])

lazyTree::(edgeFunction * )->[*]->[*]->sTree *
lazyTree edge alpha t
= sTr (suffixes t)
  where sTr [] = Leaf
        sTr ss = Branch [(a:p,sTr (y:ys)) | a<-alpha; (p,y:ys)<-[sel a
          where sel a = edge [ys | y:ys<-ss; y = a]

lazy_pst::[*]->[*]->sTree *
lazy_pst = lazyTree edge_pst

edge_pst::edgeFunction *
edge_pst [s] = (s,[[]])
edge_pst ss = ([] ,ss)

suffixes::[*]->[[*]]      || returns all non-empty suffixes
suffixes [] = []
suffixes (x:xs) = (x:xs):suffixes xs
```

By supplying different edge functions, we construct $nst(t)$ and $lst(t)$ in the same way: To construct $nst(t)$ take an edge function $edge_nst$ that labels *all* edges by a single letter. To construct $lst(t)$ take an edge function $edge_lst$ that chooses for each edge the longest common prefix of its suffix list, ignoring nested suffixes. An implementation of $edge_nst$ and $edge_lst$ can be found in [7].

³ For presentation we use the lazy functional language *MirandaTM* [16].

4.2 Ukkonen's on-line Suffix Tree Construction, Functional Version

In this section, we review Ukkonen's linear-time *on-line* suffix tree construction. The differences in our treatment compared to [18] are the following:

- While Ukkonen derives his construction in an operational style using the naive suffix tree as an intermediate step, we give a more direct and declarative presentation based on properties of suffixes.
- This approach leads to a more transparent construction, eases correctness arguments and also leads to some minor simplifications.
- It reveals the point where an implementation in a declarative language must proceed differently from Ukkonen's construction, which uses local updates to global data structures and, hence, is inherently imperative [15].

On-line construction means generating a series of suffix trees for longer and longer prefixes of the text. While $lst(\epsilon)$ is trivial (just the *root* with no edges), we study the step from $lst(t)$ to $lst(ta)$, where t is a string and a a letter. Since $lst(ta)$ must represent all ta -words, we consider all new ta -words, i.e. all ta -words, which do not occur in $lst(t)$. Every new ta -word is obviously a nonempty suffix of ta . Let sa be a new ta -word. Then $\bar{s}\bar{a}$ has to be a leaf in $lst(ta)$, since otherwise sa would be a t -word and hence occur in $lst(t)$.

If \bar{s} is a leaf in $lst(t)$, then a leaf-edge $\bar{b} \xrightarrow{w} \bar{s}$ of $lst(t)$ gives rise to a leaf-edge $\bar{b} \xrightarrow{wa} \bar{s}\bar{a}$ in $lst(ta)$. This observation led Ukkonen to the idea of representing such leaf-edges by “open” edges of the form $\bar{b} \xrightarrow{(i,\infty)} \bar{L}$, where \bar{L} denotes a leaf and (i, ∞) denotes the suffix of t starting at position i , whatever the current length of t is. Hence the suffix tree produced by the *on-line* construction will be represented as a data structure with three components:

- the global text t ,
- the global value $length = |t|$,
- the tree structure itself, with edges of the form $\bar{b} \xrightarrow{(l,r)} \bar{v}$, where the index pair (l, r) represents the edge label $t_l \dots t_{\min\{length, r\}}$.

Note that the edge label $t_l \dots t_r$ may occur several times in t , in which case the choice of (l, r) is arbitrary. The global value $length$ and the special right index value ∞ are introduced for the sake of *on-line* construction. While $length$ grows implicitly with the text, so do labels of leaf-edges. To enter a new suffix sa into the tree, nothing must be done when \bar{s} is a leaf. Hence we only have to consider the case that \bar{s} is not a leaf in $lst(t)$, or equivalently s is a nested suffix of t .

Definition 4.1 A suffix sa of ta is called relevant, if and only if s is a nested suffix of t and sa is not a t -word. \square

Now we can give an informal description of how to construct $lst(ta)$ from $lst(t)$:

- (*) Insert all relevant suffixes sa of ta into $lst(t)$.

Before we describe how to insert a relevant suffix of ta into $lst(t)$, we show that the relevant suffixes of ta form a contiguous segment of the list of all suffixes of ta , whose bounds are marked by “active suffixes”:

Definition 4.2 The active suffix of t , denoted by $\alpha(t)$, is the longest nested suffix of t .⁴ \square

Example 4.3 Consider the string $agcgacgag$ and a list of columns, where each column contains the list of all suffixes of a prefix of this string. The relevant suffixes in each column are marked by the symbol \downarrow and the active suffix is printed in bold face.

ϵ	$\downarrow a$	ag	agc	$agcg$	$agcga$	$agcgac$	$agcgacg$	$agcgacga$	$agcgacgag$
ϵ	$\downarrow g$	gc	gcf	$gcfca$	$gcfcac$	$gcfacg$	$gcfacga$	$gcfacgag$	
ϵ	$\downarrow c$	cg	cga	cga	$cgac$	$cgacg$	$cgacga$	$cgacgag$	
ϵ	\boldsymbol{g}	$\downarrow ga$	gac	$gacg$	$gacga$	$gacga$	$gacgag$		
ϵ		a	$\downarrow ac$	ac	acg	$acga$	$acgag$		
ϵ			\boldsymbol{c}	cg	\boldsymbol{cga}		$\downarrow cgag$		
ϵ				g	ga		$\downarrow gag$		
ϵ					a		\boldsymbol{ag}		
ϵ						g			
ϵ									ϵ

Lemma 4.4 For all $a \in \mathcal{A}$ and all suffixes s of t we have: sa is a relevant suffix of ta if and only if $|\alpha(t)a| \geq |sa| > |\alpha(ta)|$. \square

By Lemma 4.4 we know that the relevant suffixes of ta are “between” $\alpha(t)a$ and $\alpha(ta)$. Furthermore it is easy to show that $\alpha(ta)$ is a suffix of $\alpha(t)a$ [7]. Hence $\alpha(ta)$ is the longest suffix of $\alpha(t)a$ that is a t -word. Based on this fact we can refine algorithm (*) as follows:

- (**) Take the suffixes of $\alpha(t)a$ one after the other by decreasing length and insert them into $lst(t)$ until a suffix is found which is a t -word, and therefore equals $\alpha(ta)$.

After having explained how to find the relevant suffixes of ta , we make precise how we insert them.

Definition 4.5 Let E be an \mathcal{A}^+ -tree and s be a string that occurs in E . We call (\bar{b}, u) reference pair of s with respect to E , if \bar{b} is the root or a branching node in E and $s = bu$. If b is the longest such prefix of s , then (\bar{b}, u) is called canonical reference pair of s with respect to E . In such a case we write $\hat{s} = (\bar{b}, u)$. \square

Let sa be a relevant suffix of ta and E be the \mathcal{A}^+ -tree in which sa has to be inserted. Let $\hat{s} = (\bar{b}, u)$ and consider the following cases:

⁴ The canonical reference pair (see Definition 4.5) of an active suffix corresponds to the notion “active point” introduced in [18].

1. If \bar{s} is a node in $lst(t)$ then $u = \varepsilon$ and $\bar{s} = \bar{b}$ has no a -edge, since otherwise sa would be a t -word. Thus we only add a new open a -edge $\bar{b} \xrightarrow{(i,\infty)} \bar{L}$, where $i = length = |ta|$.
2. If \bar{s} is not a node in $lst(t)$, then $u = cw$ for some letter $c \neq a$ and some string w . Let $\bar{b} \xrightarrow{(l,r)} \bar{v}$ be a c -edge in E and let $k = l + |w|$. Then we introduce \bar{s} by splitting $\bar{b} \xrightarrow{(l,r)} \bar{v}$ into $\bar{b} \xrightarrow{(l,k)} \bar{s} \xrightarrow{(k+1,r)} \bar{v}$ and add a new open a -edge $\bar{s} \xrightarrow{(i,\infty)} \bar{L}$, where $i = length = |ta|$.

The trees resulting from 1. resp. 2. above will be denoted by $E \sqcup (\hat{s}, i)$. Putting it altogether, we can describe algorithm $(**)$ by specifying a function *update* which inserts the relevant suffixes of ta into $lst(t)$ and computes $\alpha(ta)$:

$$\begin{aligned} update(E, sa) &= (E, sa), && \text{if } sa \text{ is a } t\text{-word} \\ &= (E \sqcup (\hat{s}, i), \varepsilon), && \text{else if } s = \varepsilon \\ &= update(E \sqcup (\hat{s}, i), drop(1, sa)), \text{otherwise} \end{aligned}$$

Lemma 4.6 $update(lst(t), \alpha(t)a)$ returns the pair $(lst(ta), \alpha(ta))$. \square

Here $drop(k, w)$ denotes the string w with the first k symbols removed. There are two critical operations in this algorithm: Checking whether sa is a t -word, and splitting some edge $\bar{b} \xrightarrow{(l,r)} \bar{v}$ to introduce \bar{s} , if necessary. Both is trivial once we have computed the canonical reference pair (\bar{b}, u) for s : If $u = \varepsilon$ then sa is a t -word, if and only if \bar{b} has an a -edge. If $u = cw$ for a letter c and a string w , then there is a c -edge $\bar{b} \xrightarrow{(l,r)} \bar{v}$ and sa is a t -word, if and only if $t_{l+|u|} = a$.

The easiest way to determine (\bar{b}, u) is to follow the path for s down from the *root*, anew for each suffix s . This leads to a non-linear construction, as the length of this path can be $\mathcal{O}(n)$ in the worst case. *naiveOnline*, our functional version of Ukkonen's algorithm, uses this approach, since implementing this algorithm without local updates adds no extra overhead: Along the path from the *root* to \bar{s} , the tree may be de- and reconstructed in $\mathcal{O}(1)$ for each node visited. Thus the local update is turned into a global one with no effect on asymptotic efficiency. An implementation of *naiveOnline* is given below.

```

isTword::[*]->*>(sTree *)->bool    || if s occurs in st then
isTword [] a (Branch ts)                || isTword s a st iff sa occurs in st
  = [(c:w,st) | (c:w,st)<-ts; c = a] ~= []
isTword (x:xs) a (Branch ts)
  = w#!xs = a,                         if #xs < #w
  = isTword (drop (#w) xs) a st, otherwise
  where (c:w,st) = hd [(c:w,st) | (c:w,st)<-ts; c = x]

update::(sTree *, [*])->[*]->(sTree *, [*])  || literally from above
update (st,s) (a:as)
  = (st,s++[a]),                           if isTword s a st
  = (insRelSuff [] (a:as) st, []),          if s = []
  = update (insRelSuff s (a:as) st, drop 1 s) (a:as), otherwise

```

```

insRelSuff ::= [*] -> [*] -> (sTree *) -> (sTree *)  || insert relevant suffix
insRelSuff [] (a:as) (Branch ts)
= Branch (g ts)
  where g [] = [(a:as,Leaf)]
        g ((c:w,st):ts') = (c:w,st):g ts',           if c < a
                                = (a:as,Leaf):(c:w,st):ts', otherwise
insRelSuff (x:xs) (a:as) (Branch ts)
= Branch (g ts)
  where g ((c:w,st):ts')
        = (c:w,st):g ts',                           if c ~ x
        = (x:xs,Branch ts''):ts',                  if #xs < #w
        = (c:w,insRelSuff (drop (#w) xs) (a:as) st):ts', otherwise
          where z:zs = drop (#xs) w
                ts'' = [(a:as,Leaf),(z:zs,st)], if a < z
                = [(z:zs,st),(a:as,Leaf)], otherwise

naiveOnline ::= [*] -> (sTree *, [*])
naiveOnline t = foldl update (Branch [], []) (suffixes t)

```

4.3 McCreight's Suffix Tree Construction, Functional Version

In this section (and in section 5.3) we consider a string $t = t_1 \dots t_n$, $n \geq 2$ in which the final letter appears nowhere else in t . Let s be a suffix of t . $head_t(s)$ denotes the longest nested prefix of s , whenever $t \neq s$. Furthermore let $head_t(t) = \epsilon$. If $s = head_t(s)u$ then we denote u by $tail_t(s)$. $T_t(s)$ denotes the \mathcal{A}^+ -tree, such that w occurs in $T_t(s)$, if and only if there is a suffix s' of t , such that $|s'| \geq |s|$ and w is a prefix of s' .

The general structure of McCreight's algorithm [14] is to compute $lst(t)$ by successively inserting the suffixes s of t into the tree. Notice that the intermediate trees are not suffix trees. More precisely, given $T_t(as)$, where as is a suffix of t , the algorithm computes the canonical reference pair (\bar{h}, q) of $head_t(s)$ and the starting position j of $tail_t(s)$ and returns $T_t(s) = T_t(as) \sqcup ((\bar{h}, q), j)$. The easiest way to determine (\bar{h}, q) and j is to follow the path for s in $T_t(as)$ down from the *root*, until one “falls out of the tree” (as guaranteed by the uniqueness of the final symbol in t). This process can be described by the function *scan*:

```

scan(E, \bar{b}, i) = ((\bar{b}, \epsilon), i),      if \bar{b} has no  $t_i$ -edge in E
                    = ((\bar{b}, p), i + |p|),   else if |p| < r - l + 1
                    = scan(E, \bar{v}, i + |p|), otherwise
                      where \bar{b} \xrightarrow{(l,r)} \bar{v} is a  $t_i$ -edge in E
                            p is the longest common prefix of  $t_l \dots t_r$  and  $t_i \dots t_n$ 

```

Using *scan*, it is easy to describe how to insert a suffix of t that starts at position $i \leq n$:

```

insertSuffix(E, i) = E \sqcup ((\bar{h}, q), j)
                     where ((\bar{h}, q), j) = scan(E, root, i)

```

naiveInsertion, our functional version of McCreight's algorithm, is based on iterated use of *insertSuffix*. Again, the tree is de- and reconstructed during the scan from the root, turning the local updates into global ones without extra overhead.

```

insertSuffix::(sTree *)->[*]->sTree *
insertSuffix (Branch ts) (x:xs)
= Branch (g ts)
  where g [] = [(x:xs,Leaf)]
  g ((a:w,st):r)
    = (a:w,st):g r,      if x > a
    = (x:xs,Leaf):(a:w,st):r, if x < a
    = h bs cs:r,          otherwise
      where (as,bs,cs) = lcp w xs
      h bs [] = (a:w,st)
      h [] (c:cs) = (a:w,insertSuffix st (c:cs))
      h (b:bs) (c:cs)
        = (a:as,Branch [(c:cs,Leaf),(b:bs,st)]), if c < b
        = (a:as,Branch [(b:bs,st),(c:cs,Leaf)]), otherwise

lcp::[*]->[*]->([*],[*],[*]) || longest common prefix plus rest suffixes
lcp [] ys = ([][],[],ys)
lcp xs [] = ([][],xs,[])
lcp (x:xs) (y:ys) = (x:as,bs,cs),   if x = y
                     = ([]:x:xs,y:ys), otherwise
                     where (as,bs,cs) = lcp xs ys

naiveInsertion::[*]->sTree *
naiveInsertion t = foldl insertSuffix (Branch []) (suffixes t)

```

4.4 Asymptotic and Empirical Efficiency of the Functional Algorithms

Let $|t| = n$, and $|\mathcal{A}| = k$. The asymptotic efficiency of *naiveOnline* and *naiveInsertion* is as follows: There are $\mathcal{O}(n)$ nodes created. The path length to access each node is $\mathcal{O}(n)$ in the worst and $\mathcal{O}(\log n)$ in the expected case [2]. Selecting the suitable branch at each node introduces a factor of $\mathcal{O}(k)$. This gives a worst case of $\mathcal{O}(kn^2)$ and an expected case of $\mathcal{O}(kn \log n)$. The alphabet factor k has in fact a strong influence for large alphabets, since nodes close to the root will have close to k outgoing edges.

The asymptotic efficiency of *lazyTree* is determined by considering the number of characters read from all suffixes, and the number of operations per character read. The sum of suffix lengths is $n(n + 1)/2$. For $t = a^{n-1}\$$, all suffixes except for the longest are read to the last character. Since the functional *lazyTree* uses iteration over \mathcal{A} to group suffixes according to their first character, each character is inspected k times. This yields a tight worst case of $\mathcal{O}(kn^2)$, achieved for $a^{n-1}\$$.

The expected length of the longest repeated subword is $\mathcal{O}(\log n)$ according to [2]. Since no suffix is read beyond the point where it becomes unique, we obtain an average case efficiency of $\mathcal{O}(kn \log n)$.

Note that while *lazyTree*'s factor of k stems from the iteration over the alphabet used for grouping suffixes, for *naiveInsertion* and *naiveOnline* this factor arises from checking if an a -edge occurs in a list of $\mathcal{O}(k)$ edges.

We present some empirical results with implementations of the algorithms derived in the previous sections. All algorithms were measured on random texts (Bernoulli-distribution) over alphabets with various sizes ($k = 4, 20, 50, 90$), running on a *SPARCstation 10/40* with 32 MB. For reasons of space, we include only diagrams for $k = 50$, a sort of compromise between the amino acid alphabet ($k = 20$) and what we find in human readable ASCII files ($k \approx 90$). Measurements were done with the unix tool *rusage* and averaged over 10 runs.

Functional versions of *lazyTree*, *naiveOnline* and *naiveInsertion* were implemented in *Haskell* [5]. In contrast to *Miranda*, *Haskell* provides arrays and its compiler uses more advanced compilation techniques.⁵ We considered the following variants:

- *lazyTreeS*, *naiveOnlineS* and *naiveInsertionS*, where we represented edge labels by explicit strings (as lists of characters),
- *lazyTreeI*, *naiveOnlineI* and *naiveInsertionI*, where we represented edge labels by index pairs into the global text array, in analogy to the implementations in C.

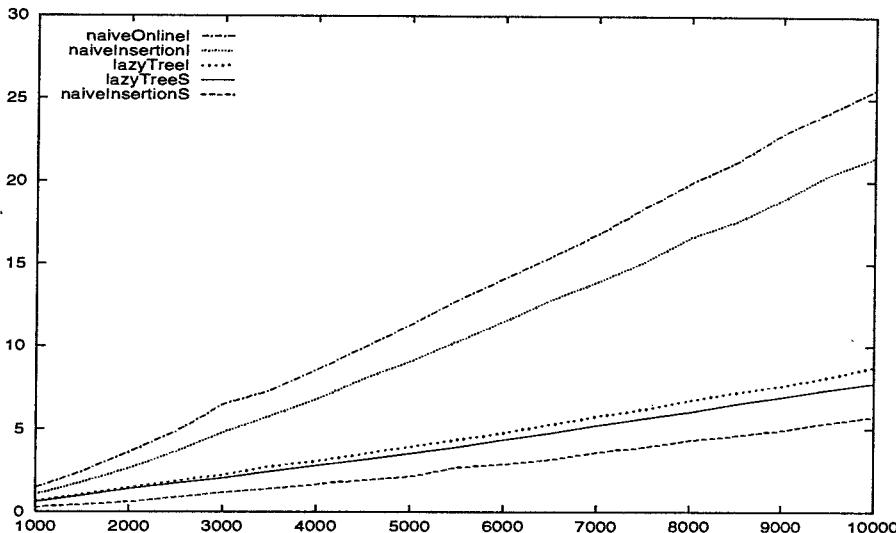
The empirical running time of *naiveOnlineS* in fact seems to be quadratic. It is not shown in the diagram below, in order to keep it readable. We attribute the poor behaviour of *naiveOnlineS* to the fact that the algorithm repeatedly appends characters to the end of the active suffix represented as a list.

From Figure 3 and related figures in [7] we obtain the following results:

- all algorithms (except *naiveOnlineS*) show close to linear behaviour.
- independent of the alphabet size, both *naiveOnlineI* and *naiveInsertionI* are slower than all the other algorithms (by a factor between 2 and 4), with *naiveInsertionI* always being the faster of the two.
- *lazyTreeS* is somewhat faster than *lazyTreeI*, with the advantage decreasing for the larger alphabets.
- *naiveInsertionS* is almost as fast as *lazyTreeS* for $k = 4$, and just beats the lazy algorithms for $k = 20, 50, 90$.

Comparative measurements of functional programs were only done up to $n = 10.000$, since some of the slower programs ran into stack size problems somewhere above. The most space-efficient program is *lazyTreeS*. However, we have not yet tuned our functional implementations with respect to space efficiency.

⁵ We used the Chalmers *Haskell* compiler. Note that there may be *Haskell* compilers that produce better code [10].

Fig. 3. Running Times of the Haskell-Programs (in seconds) for $k = 50$ 

5 Imperative Suffix Tree Algorithms

Although it was not in the original intent of this work, it turned out to be very instructive to further refine the functional algorithms into imperative programs, and redo the analysis of section 4.4. Again, the imperative implementation of *lazyTree* is new, while the imperative versions of *naiveOnline* and *naiveInsertion* are simplifications of the well-known linear-time algorithms *ukk* and *mcc*. For reasons of space, we can give only a sketch of the imperative versions.

5.1 lazyTree, Imperative Version

A careful imperative implementation of *lazyTree* is an interesting topic of its own right (see [9]). Our current version retains the basic recursion structure, uses a distribution sort for grouping suffixes according to first letters, and a naive function to determine longest common prefixes of those suffixes starting with the same letter.

5.2 Ukkonen's on-line Suffix Tree Construction, Imperative Version

We now return to the development in section 4.2 and further refine *naiveOnline* to the linear-time construction *ukk*. For achieving linear behaviour, we represent the suffix s by its canonical reference pair (\bar{b}, u) and implement a function *link* which provides direct access from (\bar{b}, u) to the representation of the longest proper suffix of s . The idea is to implement the function *link* using “suffix links” between the branching nodes.

Definition 5.1 Let E be an \mathcal{A}^+ -tree, and B be the set of its branching nodes. We define a function $f : B \setminus \{\text{root}\} \longrightarrow B$ by $f(\overline{cw}) = \overline{w}$, if and only if $(\overline{cw}, \overline{w}) \in B \setminus \{\text{root}\} \times B$. An element $(\overline{cw}, \overline{w}) \in f$ is called suffix link [14] and f is called suffix link function for E . \square

Using the suffix links to proceed from one relevant suffix to the next, we obtain *ukk*, Ukkonen's *on-line* construction. For the details of this development, see [7].

5.3 McCreight's Suffix Tree Construction, Imperative Version

We return to the development in section 4.3 and further refine *naiveInsertion* to the linear-time construction *mcc*. To get a linear algorithm (\bar{h}, q) (the canonical reference pair of $\text{head}_t(s)$) and j (the starting position of $\text{tail}_t(s)$) must be computed in constant time (averaged over all steps). McCreight's algorithm does this by exploiting the following relationships:⁶

Lemma 5.2 Let $\text{head}_t(as) = aw$ for some string w . Then

1. \overline{aw} is a branching node in $T_t(as)$,
2. w is a prefix of $\text{head}_t(s)$,
3. $w = \text{head}_t(s)$, if there is no branching node \overline{w} in $T_t(as)$. \square

To exploit these relationships McCreight's algorithm uses a representation of the “head” and the “tail” of the previous suffix and suffix links as auxiliary information. This leads to the linear-time algorithm *mcc* (details in [7]).

5.4 Asymptotic and Empirical Efficiency of the Imperative Versions

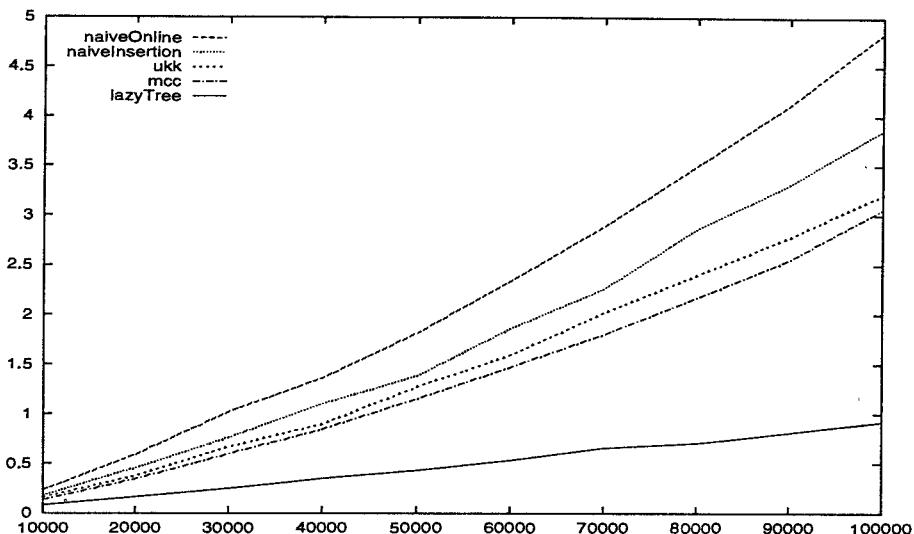
Both *ukk* and *mcc* are linear in the size of the text. Repeated traversal of lists of subtrees when a new one is added leads to a factor of k . This factor could be reduced e.g. to $\log_2 k$ by implementing subtree lists as balanced trees. With our simple tree data structure, the asymptotic efficiency is $\mathcal{O}(kn)$ for worst and expected case.

Our imperative version of *lazyTree* uses a distribution sort for grouping suffixes, avoiding iteration over the characters of the alphabet. Since the number of characters inspected is the same as in the functional implementation, we obtain a worst case of $\mathcal{O}(n^2)$ and an expected case of $\mathcal{O}(n \log n)$. Independence of alphabet size is a property shared by the suffix array algorithm of [13], whereas all other known suffix tree constructions must use more complicated data structures to reduce the alphabet factor.

⁶ McCreight explicitly uses only the second relationship, see Lemma 1 in [14].

Imperative versions of *lazyTree*, *naiveOnline*, *ukk*, *naiveInsertion* and *mcc* were implemented in C. To avoid inefficiencies by dynamic storage allocation, all C-programs represent tree nodes as elements of a statically allocated array. Measurements are shown in Figure 4 from $n = 10.000$ to $n = 100.000$.

Fig. 4. Running Times of the C-Programs (in seconds) for $k = 50$



From this and other diagrams in [7] we draw the following conclusions:

- Up to $n = 100.000$, all implementations show close to linear behaviour, irrespective of their asymptotic efficiency,
- *ukk* is always better (by a factor ≤ 3) than its naive version *naiveOnline*,
- between *mcc* and *naiveInsertion*, the same relation holds,
- *mcc* is faster than *ukk*, but the difference is not significant,
- with larger alphabets, the advantage of *ukk* and *mcc* over their naive versions decreases. This is due to the fact that the overhead of navigating through the tree is related to $\log n$.

The most interesting finding – to our own surprise – is the behaviour of *lazyTree*:

- *lazyTree*'s running time is practically linear,
- *lazyTree* is comparable to *naiveInsertion* (and half as fast as *mcc* and *ukk*) for $k = 4$,
- *lazyTree* beats all other algorithms for the larger alphabets, showing about five times the speed of the second best (*mcc*) for $k = 90$ and $n = 100.000$.

The last point is due⁷ to *lazyTree*'s independence of the alphabet size.

⁷ But only partly, since an earlier version of *lazyTree* did depend on the alphabet size, but already performed very well.

6 Conclusion

The linear-time algorithms *ukk* and *mcc* cannot – today – be implemented in a functional language while retaining their $\mathcal{O}(n)$ efficiency. This is for two reasons:

- the linear time constraint leaves no time for achieving the local updates by global reconstruction.
- previously set suffix links become obsolete by updating the tree.

However, a remedy may be on the way. The suffix tree undergoing a sequence of updates satisfies the condition of single-threadedness. No copy of an intermediate tree is used elsewhere in the program. Thus, recent ideas on adding mutable data types to functional programming languages [11], and thus incorporating local, in-place updates, apply to this case. A change of the data structure may become necessary, e.g. representing the tree by mutable arrays (which is what we use in our imperative implementation of *ukk* and *mcc*). These techniques are not yet available in today’s functional language implementations.

Let us end with some remarks on methodology. Looking at these algorithms in the two paradigms has been a very rewarding exercise. Maybe the most remarkable observations made in our experiments were the following:

- The winner in the imperative league was *lazyTree* – a new construction that was discovered in the functional paradigm, and then transliterated into the imperative world.
- At the same time, *lazyTree* was a good, but not the best competitor in the functional league. Instead, this contest was won by *naiveInsertionS*, an algorithm whose basic idea seems inherently imperative.

There is another virtue of *lazyTree*: In contrast to all other methods, it avoids random-like tree accesses during tree construction. Hence, it may be very attractive in distributed memory or database applications.

Given that *lazyTree* directly derived from an inductive definition, one may wonder why it has not been studied before. We think that this is due to the fact that in the imperative paradigm, $\mathcal{O}(n)$ solutions were known since [20]. Although it has been suggested that simpler $\mathcal{O}(n \log n)$ algorithms could do well in practise, these were usually derived from *ukk* or *mcc* (as we did with *naiveOnline* and *naiveInsertion*). Trying to avoid updates altogether was the first motivation for defining *lazyTree*. Thus, the use of functional programming has led us to this very competitive imperative algorithm.

As a side-effect of our study, we gained new insights about the close relationship between the linear-time constructions - Ukkonen’s, McCreight’s, and also Weiner’s, which has been considered sort of a mystery for 20 years. This will be explicated in [8].

References

1. A. Apostolico. The Myriad Virtues of Subword Trees. In *Combinatorial Algorithms on Words*. Springer Verlag, 1985.
2. A. Apostolico and W. Szpankowski. Self-Alignments in Words and Their Applications. *Journal of Algorithms*, 13:446–467, 1992.
3. W.I. Chang and E.L. Lawler. Approximate String Matching in Sublinear Expected Time. In *Proceedings 31st FOCS*, pages 116–124, 1990.
4. M.T. Chen and J.I. Seiferas. Efficient and Elegant Subword Tree Construction. In *Combinatorial Algorithms on Words*. Springer Verlag, 1985.
5. J.H. Fasel, P. Hudak, S. Peyton-Jones, and P. Wadler. Haskell Special Issue. *ACM SIGPLAN Notices*, 27(5), 1992.
6. R. Giegerich. Embedding Sequence Analysis in the Functional Programming Paradigm – A Feasibility Study. Report Nr. 8, Technische Fakultät, Universität Bielefeld, 1992.
7. R. Giegerich and S. Kurtz. A comparison of imperative and purely functional suffix tree constructions. Report, Technische Fakultät, Universität Bielefeld, 1994.
8. R. Giegerich and S. Kurtz. A Unifying View of Linear-Time Suffix Tree Construction. *Submitted*, 1994.
9. R. Giegerich and S. Kurtz. Implementing the Lazy Suffix Tree Construction. *In preparation*, 1994.
10. P.H. Hartel and K.G. Langendoen. Benchmarking implementations of functional languages. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture, June 1993*, pages 341–349. ACM Press, New York, NY, 1993.
11. P. Hudak. Mutable Abstract Datatypes or How to Have Your State and Munge It Too. Report, YALEU/DCS/RR-914, Yale University, Dep. of Computer Science, 1993.
12. M.E. Majster and A. Reiser. Efficient on-line Construction and Correction of Position Trees. *SIAM Journal of Computing*, 9(4):785–807, 1980.
13. U. Manber and E.W. Myers. Suffix Arrays: A New Method for On-Line String Searches. In *Proceedings of First ACM-SIAM Symposium on Discrete Algorithms*, pages 319–327, 1990.
14. E.M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2):262–272, 1976.
15. C.G. Ponder, P.C. McGeer, and A.P. Ng. Are Applicative Languages Inefficient? *SIGPLAN Notices*, 6:135–139, 1988.
16. D. Turner. An Overview of Miranda. *SIGPLAN Notices*, December 1986, 1986.
17. E. Ukkonen. Approximate String-matching with q-grams and Maximal Matches. *Theoretical Computer Science*, 92:191–211, 1992.
18. E. Ukkonen. Constructing suffix trees on-line in linear time. *Algorithms, Software, Architecture. J.v.Leeuwen (Ed.), Inform. Processing 92, Vol. I*, pages 484–492, 1992.
19. E. Ukkonen. On-line construction of suffix-trees (Revised Version of [18]). Report, A-1993-1, Dep. of Computer Science, University of Helsinki, Finland, 1993.
20. P. Weiner. Linear pattern matching algorithms. In *IEEE 14th Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.

Type classes in Haskell

Cordelia Hall, Kevin Hammond, Simon Peyton Jones
and Philip Wadler

Glasgow University

Abstract

This paper defines a set of type inference rules for resolving overloading introduced by type classes. Programs including type classes are transformed into ones which may be typed by the Hindley-Milner inference rules. In contrast to other work on type classes, the rules presented here relate directly to user programs. An innovative aspect of this work is the use of second-order lambda calculus to record type information in the program.

1. Introduction

The goal of the Haskell committee was to design a standard lazy functional language, applying existing, well-understood methods. To the committee's surprise, it emerged that there was no standard way to provide overloaded operations such as equality (`==`), arithmetic (`+`), and conversion to a string (`show`).

Languages such as Miranda¹[Tur85] and Standard ML [MTH90, MT91] offer differing solutions to these problems. The solutions differ not only between languages, but within a language. Miranda uses one technique for equality (it is defined on all types – including abstract types on which it should be undefined!), another for arithmetic (there is only one numeric type), and a third for string conversion. Standard ML uses the same technique for arithmetic and string conversion (overloading must be resolved at the point of appearance), but a different one for equality (type variables that range only over equality types).

The committee adopted a completely new technique, based on a proposal by Wadler, which extends the familiar Hindley-Milner system [Mil78] with *type classes*. Type classes provide a uniform solution to overloading, including providing operations for equality, arithmetic, and string conversion. They generalise the idea of equality types from Standard ML, and subsume the approach to string conversion used in Miranda. This system was originally

This work is supported by the SERC AQUA Project. Authors' address: Computing Science Dept, Glasgow University, 17 Lilybank Gdns., Glasgow, Scotland. Email: {cvh, kh, simonpj, wadler}@dcs.glasgow.ac.uk

¹ Miranda is a trademark of Research Software Limited.

described by Wadler and Blott [WB89, Blo91], and a similar proposal was made independently by Kae [Kae88].

The type system of Haskell is certainly its most innovative feature, and has provoked much discussion. There has been closely related work by Rouaix [Rou90] and Comack and Wright [CW90], and work directly inspired by type classes includes Nipkow and Snelting [NS91], Volpano and Smith [VS91], Jones [Jon92a, Jon93], Nipkow and Prehofer [NP93], Odersky and Läufer [OdLä91], Läufer [Läu92, Läu93], and Chen, Hudak and Odersky [CHO92].

The paper presents a source language (lambda calculus with implicit typing and with overloading) and a target language (polymorphic lambda calculus with explicit typing and without overloading). The semantics of the former is provided by translation into the latter, which has a well-known semantics [Hue 90]. Normally, one expects a theorem stating that the translation is sound, in that the translation *preserves* the meaning of programs. That is not possible here, as the translation *defines* the meaning of programs. It is a grave shortcoming of the system presented here is that there is no direct way of assigning meaning to a program, and it must be done indirectly via translation; but there appears to be no alternative. (Note, however, that [Kae88] does give a direct semantics for a slightly simpler form of overloading.)

The original type inference rules given in [WB89] were deliberately rather sparse, and were not intended to reflect the Haskell language precisely. As a result, there has been some confusion as to precisely how type classes in Haskell are defined.

1.1. Contributions of this paper. This paper spells out the precise definition of type classes in Haskell. These rules arose from a practical impetus: our attempts to build a compiler for Haskell. It presents a simplified subset of the rules we derived. The full set of rules is given elsewhere [PW91], and contains over 30 judgement forms and over 100 rules, which deal with many additional syntactic features such as type declarations, pattern matching, and list comprehensions. We have been inspired in our work by the formal semantics of Standard ML prepared by Milner, Tofte, and Harper [MTH90, MT91]. We have deliberately adopted many of the same techniques they use for mastering complexity.

This approach unites theory and practice. The industrial grade rules given here provide a useful complement to the more theoretical approaches of Wadler and Blott [WB89, Blo91], Nipkow and Snelting [NS91], Nipkow and Prehofer [NP93], and Jones [Jon92a, Jon93]. A number of simplifying assumptions made in those papers are not made here. Unlike [WB89], it is not assumed that each class has exactly one operation. Unlike [NS91], it is not assumed that the intersection of every pair of classes must be separately declared. Unlike [Jon92a], we deal directly with instance and class declarations. Each of those papers emphasises one aspect or another of the theory, while this paper stresses what we learned from practice. At the same time, these rules and the

monad-based[Wad92] implementation they support provide a clean, ‘high-level’ specification for the implementation of a typechecker, unlike more implementation oriented papers [HaBl89, Aug93, Jon92b]. A further contribution of this work is the use of explicit polymorphism in the target language.

2. Type Classes

This section introduces type classes and defines the required terminology. Some simple examples based on equality and comparison operations are introduced. Some overloaded function definitions are given and we show how they translate. The examples used here will appear as running examples through the rest of the paper.

2.1. Classes and instances. A **class** declaration provides the names and type signatures of the class *operations*:

```
class Eq a where
  (==) :: a -> a -> Bool
```

This declares that type **a** belongs to the class **Eq** if there is an operation **(==)** of type **a -> a -> Bool**. That is, **a** belongs to **Eq** if equality is defined for it.

An *instance* declaration provides a *method* that implements each class operation at a given type:

```
instance Eq Int where
  (==) = primEqInt
instance Eq Char where
  (==) = primEqChar
```

This declares that type **Int** belongs to class **Eq**, and that the implementation of equality on integers is given by **primEqInt**, which must have type **Int -> Int -> Bool**. Similarly for characters.

We can now write **2+2 == 4**, which returns **True**; or **'a' == 'b'**, which returns **False**. As usual, **x == y** abbreviates **(==) x y**. In our examples, we assume all numerals have type **Int**.

Functions that use equality may themselves be overloaded. We use the syntax **\ x ys -> e** to stand for **$\lambda x. \lambda ys. e$** .

```
member = \ x ys -> not (null ys) &&
         (x == head ys || member x (tail ys))
```

The type system infers the most general possible signature for **member**:

```
member :: (Eq a) => a -> [a] -> Bool
```

The phrase **(Eq a)** is called a *context* of the type – it limits the types that **a** can range over to those belonging to class **Eq**. As usual, **[a]** denotes the

type of lists with elements of type a. We can now inquire whether (`member 1 [2,3]`) or (`member 'a' ['c','a','t']`), but not whether (`member sin [cos,tan]`), since there is no instance of equality over functions. A similar effect is achieved in Standard ML by using equality type variables; type classes can be viewed as generalising this behaviour.

Instance declarations may themselves contain overloaded operations, if they are provided with a suitable context:

```
instance (Eq a) => Eq [a] where
  (==) = \ xs ys ->
    (null xs && null ys) ||
    (not (null xs) && not (null ys) &&
     head xs == head ys &&
     tail xs == tail ys)
```

This declares that for every type a belonging to class Eq, the type [a] also belongs to class Eq, and gives an appropriate definition for equality over lists. Note that `head xs == head ys` uses equality at type a, while `tail xs == tail ys` recursively uses equality at type [a]. We can now ask whether `['c','a','t'] == ['d','o','g']`.

Every entry in a context pairs a class name with a type variable. Pairing a class name with a type is not allowed. For example, consider the definition:

```
palindrome xs = (xs == reverse xs)
```

The inferred signature is:

```
palindrome :: (Eq a) => [a] -> Bool
```

Note that the context is (Eq a), not (Eq [a]).

2.2. Superclasses. A class declaration may include a context that specifies one or more *superclasses*:

```
class (Eq a) => Ord a where
  (<) :: a -> a -> Bool
  (=<) :: a -> a -> Bool
```

This declares that type a belongs to the class Ord if there are operations (<) and (=<) of the appropriate type, and if a belongs to class Eq. Thus, if (<) is defined on some type, then (==) must be defined on that type as well. We say that Eq is a superclass of Ord.

The superclass hierarchy must form a directed acyclic graph. An instance declaration is valid for a class only if there are also instance declarations for all its superclasses. For example

```
instance Ord Int where
  (<) = primLtInt
  (≤) = primLeInt
```

is valid, since `Eq Int` is already a declared instance.

Superclasses allow simpler signatures to be inferred. Consider the following definition, which uses both `(==)` and `(<)`:

```
search = \ x ys ->
  not (null ys) &&
  ( x == head ys || ( x < head ys &&
    search x (tail ys))
```

The inferred signature is:

```
search :: (Ord a) => a -> [a] -> Bool
```

Without superclasses, the context of the inferred signature would have been `(Eq a, Ord a)`.

2.3. Translation. The inference rules specify a translation of source programs into target programs where the overloading is made explicit.

Each instance declaration generates an appropriate corresponding *dictionary* declaration. The dictionary for a class contains dictionaries for all the superclasses, and methods for all the operators. Corresponding to the `Eq Int` and `Ord Int` instances, we have the dictionaries:

```
dictEqInt = <primEqInt>
dictOrdInt = <dictEqInt, primLtInt, primLeInt>
```

Here $\langle e_1, \dots, e_n \rangle$ builds a dictionary. The dictionary for `Ord` contains a dictionary for its superclass `Eq` and methods for `(<)` and `(≤)`.

For each operation in a class, there is a *selector* to extract the appropriate method from the corresponding dictionary. For each superclass, there is also a selector to extract the superclass dictionary from the subclass dictionary. Corresponding to the `Eq` and `Ord` classes, we have the selectors:

```
(==)      = \ (((), ==)          -> ==
getEqFromOrd = \ ((dictEq), (<, ≤)) ->
                dictEq
(<)       = \ ((dictEq), (<, ≤)) -> <
(≤)       = \ ((dictEq), (<, ≤)) -> ≤=
```

Each overloaded function has extra parameters corresponding to the required dictionaries. Here is the translation of `search`:

```
search = \ dOrd x ys ->
  not (null ys) &&
```

Type variable	α
Type constructor	χ
Class name	κ
Simple type	$\tau \rightarrow \alpha$ $\chi \tau_1 \dots \tau_k$ $(k \geq 0, k = \text{arity}(\chi))$ $\tau' \rightarrow \tau$
Overloaded type	$\rho \rightarrow \langle \kappa_1 \tau_1, \dots, \kappa_m \tau_m \rangle \Rightarrow \tau$ $(m \geq 0)$
Polymorphic type	$\sigma \rightarrow \forall \alpha_1 \dots \alpha_l. \theta \Rightarrow \tau$ $(l \geq 0)$
Context	$\theta \rightarrow \langle \kappa_1 \alpha_1, \dots, \kappa_m \alpha_m \rangle$ $(m \geq 0)$
Record Type	$\gamma \rightarrow \langle v_1 : \tau_1, \dots, v_n : \tau_n \rangle$ $(n \geq 0)$

FIGURE 1. Syntax of types

```
( (==) (getEqFromOrd dOrd) x (head ys) ||
  (<) dOrd x (head ys) &&
    search dOrd x (tail ys)))
```

Each call of an overloaded function supplies the appropriate parameters. Thus the term (**search** i [2,3]) translates to (**search dictOrdInt** i [2,3]).

If an instance declaration has a context, then its translation has parameters corresponding to the required dictionaries. Here is the translation for the instance (**Eq a**) \Rightarrow **Eq [a]**:

```
dictEqList = \ dEq ->
  (\ xs ys ->
    ( null xs && null ys ) ||
    ( not (null xs) && not (null ys) &&
      (==) dEq (head xs) (head ys) &&
      (==) (dictEqList dEq) (tail xs) (tail ys)))
```

When given a dictionary for **Eq a** this yields a dictionary for **Eq [a]**. To get a dictionary for equality on list of integers, one writes **dictEqList dictEqInt**.

The actual target language used differs from the above in that it contains extra constructs for explicit polymorphism.

3. Notation

This section introduces the syntax of types, the source language, the target language, and the various environments that appear in the type inference rules.

<i>program</i>	\rightarrow	<i>classdecls ; instdecls ; exp</i>	Programs
<i>classdecls</i>	\rightarrow	<i>classdecl₁; ... ; classdecl_n</i>	Class decls. ($n \geq 0$)
<i>instdecls</i>	\rightarrow	<i>instdecl₁; ... ; instdecl_n</i>	Instance decls. ($n \geq 0$)
<i>classdecl</i>	\rightarrow	class $\theta \Rightarrow \kappa \alpha$ where γ	Class declaration
<i>instdecl</i>	\rightarrow	instance $\theta \Rightarrow \kappa (\chi \alpha_1 \dots \alpha_k)$ where <i>binds</i>	Instance decl. ($k \geq 0$)
<i>binds</i>	\rightarrow	$\langle var_1 = exp_1 ,$ $\dots , var_n = exp_n \rangle$	($n \geq 0$)
<i>exp</i>	\rightarrow	<i>var</i>	Variable
		$\lambda var . exp$	Function abstraction
		<i>exp exp'</i>	Function application
		let <i>var</i> = <i>exp'</i> in <i>exp</i>	Local definition

FIGURE 2. Syntax of source programs

3.1. Type syntax. Figure 1 gives the syntax of types. Types come in three flavours: simple, overloaded, and polymorphic.

The record type, γ , maps class operation names to their types, and appears in the source syntax for classes. There is one subtlety. In an overloaded type ρ , entries between angle brackets may have the form $\kappa \tau$, whereas in a polymorphic type σ or a context θ entries are restricted to the form $\kappa \alpha$. The extra generality of overloaded types is required during the inference process.

3.2. Source and target syntax. Figure 2 gives the syntax of the source language and Figure 3 the syntax of the target language. We write the nonterminals of translated programs in boldface: the translated form of *var* is **var** and of *exp* is **exp**. To indicate that some target language variables and expressions represent dictionaries, we also use **dvar** and **dexp**.

The target language used here differs in that all polymorphism has been made explicit. It has constructs for type abstraction and application, and each bound variable is labeled with its type. It includes constructs to build and select from dictionaries, and to perform type abstraction and application. A program consists of a set of bindings, which may be mutually recursive, followed by an expression. The class types appearing in the translation denote monotypes; a formal translation of them appears in [HaHaPJW].

program	\rightarrow	letrec bindset in exp	Program
bindset	\rightarrow	var₁ = exp₁; ...; var_n = exp_n	Binding set ($n \geq 0$)
exp	\rightarrow	var λ pat. exp exp exp' let var = exp' in exp (exp₁, ..., exp_n) $\Lambda \alpha_1 \dots \alpha_n . exp$ exp $\tau_1 \dots \tau_n$	Variable Function abstraction Function application Local definition Dictionary formation ($n \geq 0$) Type abstraction ($n \geq 1$) Type application ($n \geq 1$)
pat	\rightarrow	var : τ (pat₁, ..., pat_n)	($n \geq 0$)

FIGURE 3. Syntax of target programs

Environment	Notation	Type
Type variable environment	AE	$\{\alpha\}$
Type constructor environment	TE	$\{\chi : k\}$
Type class environment	CE	$\{\kappa : \text{class } \theta \Rightarrow \kappa \alpha \text{ where } \gamma\}$
Instance environment	IE	$\{\text{dvar} : \forall \alpha_1 \dots \alpha_k. \theta \Rightarrow \kappa \tau\}$
Local instance environment	LIE	$\{\text{dvar} : \kappa \tau\}$
Variable environment	VE	$\{\text{var} : \sigma\}$
Environment	E	$(AE, TE, CE, IE, LIE, VE)$
Top level environment	PE	$(\{\}, TE, CE, IE, \{\}, VE)$
Declaration environment	DE	(CE, IE, VE)

FIGURE 4. Environments

3.3. Environments. The inference rules use a number of different environments, which are summarised in Figure 4. We write $ENV \ name = info$ to indicate that environment ENV maps name $name$ to information $info$. If the information is not of interest, we just write $ENV \ name$ to indicate that $name$ is in the domain of ENV . The type of a map environment is written in the symbolic form $\{name : info\}$.

We write VE of E to extract the type environment VE from the compound environment E , and similarly for other components of compound environments.

The operations \oplus and $\overset{\leftrightarrow}{\oplus}$ combine environments. The former checks that the domains of its arguments are distinct, while the latter “shadows” its left

```

 $TE_0 = \{ \text{Int}: 0,$ 
 $\quad \text{Bool}: 0,$ 
 $\quad \text{List}: 1 \}$ 

 $CE_0 = \{ \text{Eq} : \{\text{class Eq } \alpha \text{ where } \langle(==) : \alpha \rightarrow \alpha \rightarrow \text{Bool}\rangle\},$ 
 $\quad \text{Ord} : \{\text{class } \langle\text{Eq } \alpha\rangle \Rightarrow \text{Ord } \alpha$ 
 $\quad \quad \text{where } \langle(<) : \alpha \rightarrow \alpha \rightarrow \text{Bool}, (<=) : \alpha \rightarrow \alpha \rightarrow \text{Bool}\rangle\} \}$ 

 $IE_0 = \{ \text{getEqFromOrd} : \forall \alpha. \langle\text{Ord } \alpha\rangle \Rightarrow \text{Eq } \alpha,$ 
 $\quad \text{dictEqInt} : \text{Eq Int},$ 
 $\quad \text{dictEqList} : \forall \alpha. \langle\text{Eq } \alpha\rangle \Rightarrow \text{Eq } (\text{List } \alpha),$ 
 $\quad \text{dictOrdInt} : \text{Ord Int} \}$ 

 $VE_0 = \{ \langle==\rangle : \forall \alpha. \langle\text{Eq } \alpha\rangle \Rightarrow \alpha \rightarrow \alpha \rightarrow \text{Bool},$ 
 $\quad \langle<\rangle : \forall \alpha. \langle\text{Ord } \alpha\rangle \Rightarrow \alpha \rightarrow \alpha \rightarrow \text{Bool},$ 
 $\quad \langle<=\rangle : \forall \alpha. \langle\text{Ord } \alpha\rangle \Rightarrow \alpha \rightarrow \alpha \rightarrow \text{Bool} \}$ 

 $E_0 = (\{\}, TE_0, CE_0, IE_0, \{\}, VE_0)$ 

```

FIGURE 5. Initial environments

argument with its right:

$$(ENV_1 \oplus ENV_2) \ var =$$

$$\begin{cases} ENV_1 \ var & \text{if } var \in \text{dom}(ENV_1) \text{ and } var \notin \text{dom}(ENV_2) \\ ENV_2 \ var & \text{if } var \in \text{dom}(ENV_2) \text{ and } var \notin \text{dom}(ENV_1), \end{cases}$$

$$(ENV_1 \overset{\rightarrow}{\oplus} ENV_2) \ var =$$

$$\begin{cases} ENV_1 \ var & \text{if } var \in \text{dom}(ENV_1) \text{ and } var \notin \text{dom}(ENV_2) \\ ENV_2 \ var & \text{if } var \in \text{dom}(ENV_2). \end{cases}$$

For brevity, we write $E_1 \oplus E_2$ instead of a tuple of the sums of the components of E_1 and E_2 ; and we write $E \oplus VE$ to combine VE into the appropriate component of E ; and similarly for other environments. Sometimes we specify the contents of an environment explicitly and write \oplus_{ENV} .

There are three implicit side conditions associated with environments. Variables may not be declared twice in the same scope. If $E_1 \oplus E_2$ appears in a rule, then the side condition $\text{dom}(E_1) \cap \text{dom}(E_2) = \emptyset$ is implied. Every variable must appear in the environment. If $E \ var$ appears in a rule, then the side condition $var \in \text{dom}(E)$ is implied. At most one instance can be declared for a given class and given type constructor. If $IE_1 \oplus IE_2$ appears in a rule, then the side

	$E \vdash \tau$
	$E \overset{\text{over-type}}{\vdash} \theta \Rightarrow \tau$
	$E \overset{\text{poly-type}}{\vdash} \forall \alpha_1, \dots, \alpha_n. \theta \Rightarrow \tau$
TYPE-VAR	$\frac{(AE \text{ of } E) \alpha}{E \overset{\text{type}}{\vdash} \alpha}$
TYPE-CON	$\frac{(TE \text{ of } E) \chi = k \quad E \overset{\text{type}}{\vdash} \tau_i \quad (1 \leq i \leq k)}{E \overset{\text{type}}{\vdash} \chi \tau_1 \dots \tau_k}$
	$(CE \text{ of } E) \kappa_i \quad (1 \leq i \leq m)$
	$(AE \text{ of } E) \alpha_i \quad (1 \leq i \leq m)$
TYPE-PRED	$\frac{E \overset{\text{type}}{\vdash} \tau}{E \overset{\text{over-type}}{\vdash} \langle \kappa_1 \alpha_1, \dots, \kappa_m \alpha_m \rangle \Rightarrow \tau}$
TYPE-GEN	$\frac{E \oplus_{AE} \{\alpha_1, \dots, \alpha_k\} \overset{\text{over-type}}{\vdash} \theta \Rightarrow \tau}{E \overset{\text{poly-type}}{\vdash} \forall \alpha_1 \dots \alpha_k. \theta \Rightarrow \tau}$

FIGURE 6. Rules for types

condition

$$\begin{aligned} \forall \kappa_1 (\chi_1 \alpha_1 \dots \alpha_m) &\in IE_1. \\ \forall \kappa_2 (\chi_2 \alpha_1 \dots \alpha_n) &\in IE_2. \\ \kappa_1 &\neq \kappa_2 \vee \chi_1 \neq \chi_2 \end{aligned}$$

is implied.

In some rules, types in the source syntax constrain the environments generated from them. This is stated explicitly by the *determines* relation, defined as:

$$\tau \text{ determines } AE \iff ftv(\tau) = AE$$

$$\theta \text{ determines } LIE \iff \theta = ran(LIE)$$

4. Rules

This section gives the inference rules for the various constructs in the source language. We consider in turn types, expressions, dictionaries, class declarations, instance declarations, and full programs.

4.1. Types. The rules for types are shown in Figure 6. The three judgement forms defined are summarised in the upper left corner. A judgement of the form

$$E \stackrel{\text{type}}{\vdash} \tau$$

holds if in environment E the simple type τ is valid. In particular, all type variables in τ must appear in AE of E (as checked by rule TYPE-VAR), and all type constructors in τ must appear in TE of E with the appropriate arity (as checked by rule TYPE-CON). The other judgements act similarly for overloaded types and polymorphic types.

4.2. Expressions. The rules for expressions are shown in Figure 7. A judgement of the form

$$E \stackrel{\text{exp}}{\vdash} exp : \tau \rightsquigarrow \text{exp}$$

holds if in environment E the expression exp has simple type τ and yields the translation exp . The other two judgements act similarly for overloaded and polymorphic types.

4.3. Dictionaries. The inference rules for dictionaries are shown in Figure 8. A judgement of the form

$$E \stackrel{\text{dict}}{\vdash} \kappa \tau \rightsquigarrow \text{dexp}$$

holds if in environment E there is an instance of class κ at type τ given by the dictionary dexp . The other two judgements act similarly for overloaded and polymorphic instances.

4.4. Class declarations. The rule for class declarations is given in Figure 9. Although the rule looks formidable, its workings are straightforward.

A judgement of the form

$$PE \stackrel{\text{classdecl}}{\vdash} \text{classdecl} : DE \rightsquigarrow \text{bindset}$$

holds if in environment PE the class declaration classdecl is valid, generating new environment DE and yielding translation bindset . In the compound environment $DE = (CE, IE, VE)$, the class environment CE has one entry that describes the class itself, the instance environment IE has one entry for each superclass of the class (given the class dictionary, it selects the appropriate superclass dictionary) and the value environment VE has one entry for each operator of the class (given the class dictionary, it selects the appropriate method).

$E \vdash \text{exp} : \tau \rightsquigarrow \text{exp}$
$E \vdash \text{over-exp} \text{exp} : \rho \rightsquigarrow \text{exp}$
$E \vdash \text{poly-exp} \text{exp} : \sigma \rightsquigarrow \text{exp}$
$\text{TAUT } \frac{(VE \text{ of } E) \text{ var} = \sigma}{E \vdash \text{var} : \sigma \rightsquigarrow \text{var}}$
$\text{SPEC } \frac{\begin{array}{c} E \vdash \text{poly-exp} \text{var} : \forall \alpha_1 \dots \alpha_k. \theta \Rightarrow \tau \rightsquigarrow \text{var} \\ E \vdash \text{type} \tau_i \end{array}}{E \vdash \text{var} : (\theta \Rightarrow \tau)[\tau_1/\alpha_1, \dots, \tau_k/\alpha_k] \rightsquigarrow \text{var } \tau_1 \dots \tau_k} \quad (1 \leq i \leq k)$
$\text{REL } \frac{\begin{array}{c} E \vdash \text{over-exp} \text{var} : \theta \Rightarrow \tau \rightsquigarrow \text{exp} \\ E \vdash \text{dicts} \theta \rightsquigarrow \text{dexprs} \end{array}}{E \vdash \text{var} : \tau \rightsquigarrow \text{exp dexprs}}$
$\text{ABS } \frac{\begin{array}{c} E \xrightarrow{\vec{\oplus}_{VE}} \{\text{var} : \tau'\} \vdash \text{exp} : \tau \rightsquigarrow \text{exp} \\ E \vdash \text{exp} \lambda \text{var}. \text{exp} : \tau' \rightarrow \tau \rightsquigarrow \lambda \text{var} : \tau'. \text{exp} \end{array}}{E \vdash \lambda \text{var}. \text{exp} : \tau' \rightarrow \tau \rightsquigarrow \lambda \text{var} : \tau'. \text{exp}}$
$\text{COMB } \frac{\begin{array}{c} E \vdash \text{exp} : \tau' \rightarrow \tau \rightsquigarrow \text{exp} \\ E \vdash \text{exp}' : \tau' \rightsquigarrow \text{exp}' \end{array}}{E \vdash (\text{exp exp}') : \tau \rightsquigarrow (\text{exp exp}')}}$
$\text{PRED } \frac{\begin{array}{c} E \oplus \text{LIE} \vdash \theta \rightsquigarrow \text{dpat} \\ E \oplus \text{LIE} \vdash \text{exp} : \tau \rightsquigarrow \text{exp} \end{array}}{E \vdash \text{over-exp} \text{exp} : \theta \Rightarrow \tau \rightsquigarrow \lambda \text{dpat} : \theta. \text{exp}} \quad \theta \text{ determines LIE}$
$\text{GEN } \frac{\begin{array}{c} E \oplus_{AE} \{\alpha_1, \dots, \alpha_k\} \vdash \text{exp} : \theta \Rightarrow \tau \rightsquigarrow \text{exp} \\ E \vdash \text{poly-exp} \text{exp} : \forall \alpha_1 \dots \alpha_k. \theta \Rightarrow \tau \rightsquigarrow \Lambda \alpha_1 \dots \alpha_k. \text{exp} \end{array}}{E \vdash \text{poly-exp} \text{exp} : \forall \alpha_1 \dots \alpha_k. \theta \Rightarrow \tau \rightsquigarrow \Lambda \alpha_1 \dots \alpha_k. \text{exp}}$
$\text{LET } \frac{\begin{array}{c} E \vdash \text{poly-exp} \text{exp}' : \sigma \rightsquigarrow \text{exp}' \\ E \xrightarrow{\vec{\oplus}_{VE}} \{\text{var} : \sigma\} \vdash \text{exp} : \tau \rightsquigarrow \text{exp} \end{array}}{E \vdash \text{exp} \text{let var} = \text{exp}' \text{ in exp} : \tau \rightsquigarrow \text{let var} = \text{exp}' \text{ in exp}}$

FIGURE 7. Rules for expressions

$E \stackrel{\text{dict}}{\vdash} \kappa \tau \rightsquigarrow \mathbf{dexp}$	
$E \stackrel{\text{over-dict}}{\vdash} \theta \Rightarrow \kappa \tau \rightsquigarrow \mathbf{dexp}$	
$E \stackrel{\text{poly-dict}}{\vdash} \forall \alpha_1 \dots \alpha_n. \theta \Rightarrow \kappa \tau \rightsquigarrow \mathbf{dexp}$	
$E \stackrel{\text{dicts}}{\vdash} \theta \rightsquigarrow \mathbf{dexps}$	
$\text{DICT-TAUT-LIE } \frac{(LIE \text{ of } E) \text{ dvar} = \kappa \alpha}{E \stackrel{\text{dict}}{\vdash} \kappa \alpha \rightsquigarrow \text{dvar}}$	
$\text{DICT-TAUT-IE } \frac{(IE \text{ of } E) \text{ dvar} = \forall \alpha_1 \dots \alpha_n. \theta \Rightarrow \kappa (\chi \alpha_1 \dots \alpha_n)}{E \stackrel{\text{poly-dict}}{\vdash} \forall \alpha_1 \dots \alpha_n. \theta \Rightarrow \kappa (\chi \alpha_1 \dots \alpha_n) \rightsquigarrow \text{dvar}}$	
$\text{DICT-SPEC } \frac{E \stackrel{\text{poly-dict}}{\vdash} \forall \alpha_1 \dots \alpha_n. \theta \Rightarrow \kappa \tau \rightsquigarrow \mathbf{dexp}}{E \stackrel{\text{over-dict}}{\vdash} (\theta \Rightarrow \kappa \tau)[\tau_1/\alpha_1, \dots, \tau_n/\alpha_n] \rightsquigarrow \mathbf{dexp} \tau_1 \dots \tau_n}$	
$\text{DICT-REL } \frac{\begin{array}{l} E \stackrel{\text{over-dict}}{\vdash} \theta \Rightarrow \kappa \tau \rightsquigarrow \mathbf{dexp} \\ E \stackrel{\text{dicts}}{\vdash} \theta \rightsquigarrow \mathbf{dexps} \end{array}}{E \stackrel{\text{dict}}{\vdash} \kappa \tau \rightsquigarrow \mathbf{dexp} \mathbf{dexps}}$	
$\text{DICTS } \frac{E \stackrel{\text{dict}}{\vdash} \kappa_i \tau_i \rightsquigarrow \mathbf{dexp}_i \quad (1 \leq i \leq n)}{E \stackrel{\text{dicts}}{\vdash} \langle \kappa_1 \tau_1, \dots, \kappa_n \tau_n \rangle \rightsquigarrow \langle \mathbf{dexp}_1, \dots, \mathbf{dexp}_n \rangle}$	

FIGURE 8. Rules for dictionaries

4.5. Instance declarations. The rule for instance declarations is given in Figure 11. Again the rule looks formidable, and again its workings are straightforward.

A judgement of the form

$$PE \stackrel{\text{instdecl}}{\vdash} \text{instdecl} : IE \rightsquigarrow \text{bindset}$$

holds if in environment PE the instance declaration instdecl is valid, generating new environment IE and yielding translation bindset . The instance environment IE contains a single entry corresponding to the instance declaration, and the bindset contains a single binding. If the header of the instance declaration is $\theta \Rightarrow \kappa \tau$, then the corresponding instance is a function that expects one dictionary for each entry in θ , and returns a dictionary for the instance.

$E \vdash \text{classdecl} : DE \rightsquigarrow \text{bindset}$	
	$PE \oplus AE \stackrel{\text{type}}{\vdash} \alpha \quad \alpha \text{ determines } AE$
	$PE \oplus AE \oplus LIE \stackrel{\text{dicts}}{\vdash} \theta \rightsquigarrow \text{dpat} \quad \theta \text{ determines } LIE$
	$PE \oplus AE \stackrel{\text{sig}}{\vdash} \gamma \rightsquigarrow \text{mpat}$
	$\text{pat} = (\text{dpat}, \text{mpat}) : (PE(\theta), PE(\gamma))$
	$CLASS \frac{\text{classdecl}}{PE \vdash \text{class } \theta \Rightarrow \kappa \alpha \text{ where } \gamma}$
	$:$
	$(\{\kappa : \text{class } \theta \Rightarrow \kappa \alpha \text{ where } \gamma\},$
	$\{\text{dvar} : \Lambda \alpha. \langle \kappa \alpha \rangle \Rightarrow \kappa' \tau'$
	$ \text{ dvar} : \kappa' \tau' \in LIE\},$
	$\{\text{var} : \Lambda \alpha. \langle \kappa \alpha \rangle \Rightarrow \tau \text{ var} : \tau \in \gamma\})$
	\rightsquigarrow
	$\{\text{dvar} = \Lambda \alpha. \lambda \text{ pat} . \text{dvar}$
	$ \text{ dvar} \in \text{dom}(LIE)\} \cup$
	$\{\text{var} = \Lambda \alpha. \lambda \text{ pat} . \text{var} \text{ var} \in \text{dom}(\gamma)\}$

FIGURE 9. Rule for class declarations

$E \stackrel{\text{sig}}{\vdash} \text{sig} \rightsquigarrow \text{sig}$	
	$SIGS \frac{E \stackrel{\text{type}}{\vdash} \tau_i \quad (1 \leq i \leq m)}{E \stackrel{\text{sig}}{\vdash} \langle \text{var}_1 : \tau_1, \dots, \text{var}_m : \tau_m \rangle \rightsquigarrow \langle \text{var}_1, \dots, \text{var}_m \rangle}$

FIGURE 10. Rule for class signatures

4.6. Programs. We omit the rules for declaration sequences and programs; these appear in the full technical report [HaHaPJW].

5. Conclusions

This paper presents a minimal, readable set of inference rules to handle type classes in Haskell, derived from the full static semantics [PW91]. An important feature of this style of presentation is that it scales up well to a description of the entire Haskell language. We have found in practice that these rules can be directly implemented using monads. This style has been applied to the full static semantics in order to construct the type checker used in the Glasgow Haskell compiler, as well as virtually all other passes in the compiler. It has undoubtedly saved us from initially making countless bookkeeping errors, and

$E \vdash \text{instdecl} : IE \rightsquigarrow \text{bindset}$
$(CE \text{ of } PE) \kappa = \text{class } \theta' \Rightarrow \kappa \alpha \text{ where } \gamma'$
$PE \oplus AE \stackrel{\text{type}}{\vdash} \tau \quad \tau \text{ determines } AE$
$PE \oplus AE \oplus LIE \stackrel{\text{dicts}}{\vdash} \theta \rightsquigarrow \text{dpat} \quad \theta \text{ determines } LIE$
$PE \oplus AE \oplus LIE \stackrel{\text{dicts}}{\vdash} \theta'[\tau/\alpha] \rightsquigarrow \text{dexp}$
$\text{INST} \quad \frac{PE \oplus AE \oplus LIE \stackrel{\text{binds}}{\vdash} \text{binds} : \gamma'[\tau/\alpha] \rightsquigarrow \text{binds}}{PE \vdash \text{instance } \theta \Rightarrow \kappa \tau \text{ where binds}}$
\vdots
$\{ \text{dvar} = \forall \text{ dom}(AE). \theta \Rightarrow \kappa \tau \}$
\rightsquigarrow
$\text{dvar} = \Lambda \text{ dom}(AE). \lambda \text{ dpat} : PE(\theta). \langle \text{dexp}, \text{binds} \rangle$

FIGURE 11. Rule for instance declarations

$E \stackrel{\text{binds}}{\vdash} \text{binds} : \gamma \rightsquigarrow \text{binds}$
$\text{BINDS} \quad \frac{E \stackrel{\text{exp}}{\vdash} exp_i : \text{exp}_i \rightsquigarrow \tau_i \quad (1 \leq i \leq m)}{E \stackrel{\text{binds}}{\vdash} \langle var_1 = exp_1, \dots, var_m = exp_m \rangle}$
\vdots
$\langle var_1 : \tau_1, \dots, var_m : \tau_m \rangle$
\rightsquigarrow
$\langle \text{var}_1 = \text{exp}_1, \dots, \text{var}_m = \text{exp}_m \rangle$

FIGURE 12. Rule for instance bindings

continues to pay off as we maintain our code and train students to work with it.

References

- [Aug93] L. Augustsson, Implementing Haskell Overloading. In *Functional Programming and Computer Architecture*, Copenhagen, June 1993.
- [Blo91] S. Blott, *Type classes*. Ph.D. Thesis, Glasgow University, 1991.
- [CHO92] K. Chen, P. Hudak, and M. Odersky, Parametric Type Classes. In *Lisp and Functional Programming*, 1992, pp. 170–181.
- [CW90] G. V. Comack and A. K. Wright, Type dependent parameter inference. In *Programming Language Design and Implementation*, White Plains, New York, June 1990, ACM Press.

- [HaBl89] K. Hammond and S. Blott, Implementing Haskell Type Classes. In *1989 Glasgow Workshop on Functional Programming*, Fraserburgh, Scotland, September 1989, Springer-Verlag WICS, pp. 266–286.
- [HaHaPJW] C.V. Hall, K. Hammond, S.L.Peyton Jones and P. Wadler, Type Classes In Haskell. Department of Computing Science, Glasgow University, Jan 1994.
- [Hue 90] Gerard Huet, editor, *Logical Foundations of Functional Programming*, Addison Wesley, 1990. See Part II, Polymorphic Lambda Calculus, especially the introduction by Reynolds.
- [Jon92a] M. P. Jones, A theory of qualified types. In *European Symposium on Programming*, Rennes, February 1992, LNCS 582, Springer-Verlag.
- [Jon92b] M. P. Jones, Efficient Implementation of Type Class Overloading. Dept. of Computing Science, Oxford University.
- [Jon93] M. P. Jones, A System of Constructor Classes: Overloading and Implicit Higher-Order Polymorphism. In *Functional Programming and Computer Architecture*, Copenhagen, June 1993, pp. 52–61.
- [Kae88] S. Kael, Parametric polymorphism. In *European Symposium on Programming*, Nancy, France, March 1988, LNCS 300, Springer-Verlag.
- [Läu92] Polymorphic Type Inference and Abstract Data Types. K. Läufer, Ph.D. Thesis, New York University, 1992.
- [Läu93] An Extension of Haskell with First-Class Abstract Types. K. Läufer, Technical Report, Loyola University of Chicago, 1993.
- [MTH90] R. Milner, M. Tofte, and R. Harper, *The definition of Standard ML*, MIT Press, Cambridge, Massachusetts, 1990.
- [MT91] R. Milner and M. Tofte, *Commentary on Standard ML*, MIT Press, Cambridge, Massachusetts, 1991.
- [Mil78] R. Milner, A theory of type polymorphism in programming. *J. Comput. Syst. Sci.* 17, 1978, pp. 348–375.
- [NP93] T. Nipkow and C. Prehofer, Type Checking Type Classes. In *ACM Symposium on Principles of Programming Languages*, January 1993, pp. 409–418.
- [NS91] T. Nipkow and G. Snelting, Type Classes and Overloading Resolution via Order-Sorted Unification. In *Functional Programming Languages and Computer Architecture*, Boston, August 1991, LNCS 523, Springer-Verlag.
- [OdLä91] M. Odersky and K. Läufer, Type classes are signatures of abstract types. Technical Report, IBM TJ Watson Research Centre, May 1991.
- [PW91] S. L. Peyton Jones and P. Wadler, A static semantics for Haskell. Department of Computing Science, Glasgow University, May 1991.
- [Rou90] F. Rouaix, Safe run-time overloading. In *ACM Symposium on Principles of Programming Languages*, San Francisco, January 1990, ACM Press.
- [Tur85] D. A. Turner, Miranda: A non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture*, Nancy, France, September 1985, LNCS 201, Springer-Verlag, pp. 1–16.
- [VS91] D. M. Volpano and G. S. Smith, On the complexity of ML typability with overloading. In *Functional Programming Languages and Computer Architecture*, Boston, August 1991, LNCS 523, Springer-Verlag.
- [Wad92] P. L. Wadler, The essence of functional programming. In *ACM Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, January 1992.
- [WB89] P. L. Wadler and S. Blott, How to make ad-hoc polymorphism less ad hoc, In *ACM Symposium on Principles of Programming Languages*, Austin, Texas, January 1989, pp. 60–76.

Lazy type inference for the strictness analysis of lists

Chris Hankin¹ and Daniel Le Métayer²

¹ Department of Computing, Imperial College, LONDON SW7 2BZ, UK

² INRIA/IRISA, Campus de Beaulieu, 35042 RENNES CEDEX, FRANCE

Abstract. We present a type inference system for the strictness analysis of lists and we show that it can be used as the basis for an efficient algorithm. The algorithm is as accurate as the usual abstract interpretation technique. One distinctive advantage of this approach is that it is not necessary to impose an abstract domain of a particular depth prior to the analysis: the lazy type algorithm will instead explore the part of a potentially infinite domain required to prove the strictness property.

1 Introduction

Simple strictness analysis returns information about the fact that the result of a function application is undefined when some of the arguments are undefined. This information can be used in a compiler for a lazy functional language because the argument of a strict function can be evaluated (up to weak head normal form) and passed by value. However a more sophisticated property might be useful in the presence of lists or other recursive data structures which are pervasive in functional programs. For example, consider the following program:

$$\begin{array}{ll} \text{sum } \mathbf{nil} & = 0 \\ \text{sum } \mathbf{cons}(x, xs) & = x + (\text{sum } xs) \\ \text{append } \mathbf{nil} \ l & = l \\ \text{append } \mathbf{cons}(x, xs) \ l & = \mathbf{cons}(x, (\text{append } xs \ l)) \\ H \ l_1 \ l_2 & = \text{sum}(\text{append } l_1 \ l_2) \end{array}$$

Rather than suspending the evaluation of each recursive call to *append* and returning the weak head normal form $\mathbf{cons}(x, (\text{append } xs \ l))$, we may want to compute directly the normal form of the argument to *sum* in *H* because the whole list will be needed. There have been a number of proposals to extend strictness analysis to recursively defined data structures [4, 21, 25, 26]. These have led to sophisticated analyses but two aspects of the problem have received little attention until recently: (1) the integration of the results of the analysis into a real compiler, (2) the efficiency of the algorithm implementing the analysis. The first issue has been tackled recently both from an experimental point of view [11, 16] and from a theoretical point of view [5, 8]. We are concerned with the second issue in this paper. The abstract interpretation and the projections approaches have led to the construction of analyses based on rich domains which

make them intractable even for some simple examples. Techniques striving for a better representation of the domains do not really solve the problem [12, 17].

This observation has motivated some researchers, [2, 18, 19, 20], to develop non-standard type inference systems for strictness analysis. Kuo and Mishra, [20], proposed a type inference system for strictness information; they developed a sound and complete inference algorithm but did not show the correctness of the inference system (with respect to a standard semantics). In [21] it is shown how their type inference system can be extended to a form of full strictness for lists and to the 4-point domain of Wadler [25].

The other authors, [2, 18, 19], have developed sound and complete inference systems but have not given much attention to algorithms. In [13] we demonstrated a technique for deriving efficient static analysis algorithms from type inference systems. The basis for this work was Jensen's conjunctive strictness logic [18]; we used techniques similar to [14, 15] to refine the logic into an algorithm.

In this paper, we follow the approach taken in [13] to construct an efficient algorithm for the analysis of lists. The algorithm is both correct and complete with respect to the usual abstract interpretation approach. So it does not incur the loss of accuracy of previous type inference systems for the analysis of lists [21]. The core of the algorithm is the notion of lazy types (or lazily evaluated types) which allows us to compute only the information required to answer a particular question about the strictness of a function. One significant advantage of the approach is that it extends naturally to domains of any depth and domains are only explored at the particular depth required for the original question. In other words, we do not have to choose a particular domain before the analysis as is usually done for abstract interpretation (except when widening operators are used as in [7]).

We first describe an extension of Jensen's strictness logic [18] to include an analogue of Wadler's 4-point domain [25] (Section 2). In Section 3, we introduce the notion of lazy types for lists and we present the corresponding system. We state the correctness and completeness properties with respect to the original system and we proceed in Section 4 to define the lazy type inference algorithm. The algorithm can in fact be derived in the same way as in [13] and the correctness proof follows for the same reasons. Section 5 is an example of the functioning of the algorithm. We show in Section 6 that the type system and algorithm can be extended to domains of unbounded depth and we present an example showing that the algorithm naturally explores the depth of the domain required by a particular question. Related work is discussed in Section 7.

2 A strictness logic for the analysis of lists

We consider a strongly typed language, Λ_L , with terms defined by the following syntax:

$$\begin{aligned} e = & \ x \mid c \mid \lambda x.e \mid e_1 e_2 \mid \text{fix}(\lambda g.e) \mid \text{cond}(e_1, e_2, e_3) \mid \\ & \text{nil} \mid \text{cons}(e_1, e_2) \mid \text{hd}(e) \mid \text{tl}(e) \mid \text{case}(e_1, e_2, e_3) \end{aligned}$$

The **case** operator is used in the translation of pattern matching. For example, the *sum* function from the previous section is translated as:

$$\text{sum}(l) = \text{case}(0, f, l) \quad \text{where } f x xs = x + (\text{sum } xs)$$

The loss of accuracy that occurs without the **case** operator is discussed in [25].

Abstract interpretation represents the strictness properties of a function by an abstract function defined on boolean domains [23]. For instance $g_{abs} t f = f$ means that g is undefined if its second argument is undefined. In terms of types, this property is represented by $g : t \rightarrow f \rightarrow f$. Notice that t and f are now (non-standard) types. Conjunctive types are required to retain the power of abstract interpretation: a strict function like $+$ must have type $(f \rightarrow t \rightarrow f) \wedge (t \rightarrow f \rightarrow f)$. Let us now turn to the types used for the representation of properties of lists. As a first stage, we consider the extension of the boolean domain to Wadler's 4-point domain [25]. We show that this extension can be generalised to domains of unbounded depth in Section 6. The four elements of the domain are $f \leq \infty \leq f_\epsilon \leq t$ where ∞ represents infinite lists or lists ending with an undefined element and f_ϵ corresponds to finite lists whose elements may be undefined (plus the lists represented by ∞).

The ordering on types is described in Fig. 1. We define $=$ as the equivalence induced by the ordering on types: $\sigma = \tau \Leftrightarrow \sigma \leq \tau$ and $\tau \leq \sigma$. The type inference system is shown in Fig. 2. Γ is an environment mapping variables to formulae (i.e. strictness types). In the rule **Cond-1**, σ represents the standard type of e_2 (or e_3). This system is an extension of [13, 18] and the soundness and completeness proofs of the logic (with respect to traditional abstract interpretation) follow straightforwardly from [19]. As an illustration, we show how the property, $\text{sum} : f_\epsilon \rightarrow f$, can be derived in this logic:

$$\begin{array}{c}
 \begin{array}{ccc}
 & A & B \\
 \text{Conj} & \frac{}{[s : f_\epsilon \rightarrow f, l : f_\epsilon] \vdash \lambda x. \lambda x s. x + (s xs) : t \rightarrow f_\epsilon \rightarrow f \wedge f \rightarrow t \rightarrow f} & C \\
 \text{Case - 3} & \frac{}{[s : f_\epsilon \rightarrow f, l : f_\epsilon] \vdash \text{case}(0, \lambda x. \lambda x s. x + (s xs), l) : f} & \\
 & \vdots & \\
 \text{Abs} & \frac{}{\vdash (\lambda s. \lambda l. \text{case}(0, \lambda x. \lambda x s. x + (s xs), l)) : (f_\epsilon \rightarrow f) \rightarrow (f_\epsilon \rightarrow f)} & \\
 \text{Fix} & \frac{\vdash \text{fix}(\lambda s. \lambda l. \text{case}(0, \lambda x. \lambda x s. x + (s xs), l)) : f_\epsilon \rightarrow f}{\vdash \text{sum} : f_\epsilon \rightarrow f} & \\
 & \vdots &
 \end{array}
 \end{array}$$

where A is:

$$\begin{array}{c}
 \vdots \\
 \frac{}{[s : f_\epsilon \rightarrow f, l : f_\epsilon, x : t, xs : f_\epsilon] \vdash x + (s xs) : f} \\
 \vdots \\
 \text{Abs } \frac{}{[s : f_\epsilon \rightarrow f, l : f_\epsilon] \vdash \lambda x. \lambda x s. x + (s xs) : t \rightarrow f_\epsilon \rightarrow f}
 \end{array}$$

B is:

$$\begin{array}{c}
 \vdots \\
 \frac{}{[s : f_\epsilon \rightarrow f, l : f_\epsilon, x : f, xs : t] \vdash x + (s xs) : f} \\
 \vdots \\
 \text{Abs } \frac{}{[s : f_\epsilon \rightarrow f, l : f_\epsilon] \vdash \lambda x. \lambda x s. x + (s xs) : f \rightarrow t \rightarrow f}
 \end{array}$$

and C is:

$$\mathbf{Var} [s : \mathbf{f}_\epsilon \rightarrow \mathbf{f}, l : \mathbf{f}_\epsilon] \vdash l : \mathbf{f}_\epsilon$$

Note that A and B make use of the implicit assumption about the type of +. Any environment is supposed to contain all the types of primitive operators.

$\mathbf{f} \leq \phi$	$\phi \leq \phi$	$\infty \leq \mathbf{f}_\epsilon$	$\phi \leq t$	$t_{\sigma \rightarrow \tau} \leq t_\sigma \rightarrow t_\tau$
$\frac{\phi \leq \psi, \psi \leq \chi}{\phi \leq \chi}$	$\frac{\phi \leq \psi_1, \phi \leq \psi_2}{\phi \leq \psi_1 \wedge \psi_2}$		$\phi \wedge \psi \leq \phi$	$\phi \wedge \psi \leq \psi$
			$\frac{\phi' \leq \phi, \psi \leq \psi'}{\phi \rightarrow \psi \leq \phi' \rightarrow \psi'}$	

Fig. 1. The ordering on types

3 Lazy Types

We introduce a slightly restricted language of strictness formulae T_I (Fig. 3); this language is closely related to van Bakel's strict types [1]. Basically strict types do not allow intersections on the right hand side of an arrow. This restriction is convenient because it does not weaken the expressive power of the system and it makes type manipulation easier.

We then define the notion of *complete type*. The restriction to complete types allows us to avoid the use of weakening because a complete type contains (is the conjunction of) all of the elements greater than (or equal to) it.

Definition 1.

$$\begin{aligned} CT(\tau) &= \bigwedge \{ct(\sigma) \mid \sigma \in Sup(\tau)\} \\ Sup(\sigma) &= \{\sigma' \in T_S \mid \sigma' \geq \sigma\} \\ ct(t) &= t \quad ct(\mathbf{f}) = \mathbf{f} \quad ct(\infty) = \infty \quad ct(\mathbf{f}_\epsilon) = \mathbf{f}_\epsilon \\ ct(\sigma \wedge \tau) &= ct(\sigma) \wedge ct(\tau) \\ ct(\sigma \rightarrow \tau) &= CT(\sigma) \rightarrow ct(\tau) \end{aligned}$$

$Sup(\sigma)$ can be defined by induction on σ . Notice that CT can be extended to contexts in the obvious way.

Finally, we can define the notion of *most general type* of an expression (with respect to some context): it is the conjunction of all of the types possessed by the expression in the given environment.

Definition 2 (Most General Types).

$$MGT(\Gamma, e) = CT(\bigwedge \{\sigma_i \in T_S \mid \Gamma \vdash_T e : \sigma_i\})$$

Conj	$\frac{\Gamma \vdash_T e : \psi_1 \quad \Gamma \vdash_T e : \psi_2}{\Gamma \vdash_T e : \psi_1 \wedge \psi_2}$	Weak	$\frac{\Gamma \leq \Delta \quad \Delta \vdash_T e : \phi \quad \phi \leq \psi}{\Gamma \vdash_T e : \psi}$
Var	$\Gamma[x \mapsto \phi] \vdash_T x : \phi$	Abs	$\frac{\Gamma[x \mapsto \phi] \vdash_T e : \psi}{\Gamma \vdash_T \lambda x. e : (\phi \rightarrow \psi)}$
App	$\frac{\Gamma \vdash_T e_1 : (\phi \rightarrow \psi) \quad \Gamma \vdash_T e_2 : \phi}{\Gamma \vdash_T e_1 e_2 : \psi}$	Fix	$\frac{\Gamma \vdash_T (\lambda g. e) : \phi \rightarrow \phi}{\Gamma \vdash_T \text{fix}(\lambda g. e) : \phi}$
Cond-1	$\frac{\Gamma \vdash_T e_1 : \mathbf{f}}{\Gamma \vdash_T \text{cond}(e_1, e_2, e_3) : \mathbf{f}_\sigma}$	Cond-2	$\frac{\Gamma \vdash_T e_2 : \phi \quad \Gamma \vdash_T e_3 : \phi}{\Gamma \vdash_T \text{cond}(e_1, e_2, e_3) : \phi}$
Hd	$\frac{\Gamma \vdash_T e : \mathbf{f}}{\Gamma \vdash_T \text{hd}(e) : \mathbf{f}}$	Tl-1	$\frac{\Gamma \vdash_T e : \mathbf{f}}{\Gamma \vdash_T \text{tl}(e) : \mathbf{f}}$
		Tl-2	$\frac{\Gamma \vdash_T e : \infty}{\Gamma \vdash_T \text{tl}(e) : \infty}$
		Cons-1	$\frac{\Gamma \vdash_T e_2 : \infty}{\Gamma \vdash_T \text{cons}(e_1, e_2) : \infty}$
Cons-2	$\frac{\Gamma \vdash_T e_2 : \mathbf{f}_\epsilon}{\Gamma \vdash_T \text{cons}(e_1, e_2) : \mathbf{f}_\epsilon}$	Cons-3	$\frac{\Gamma \vdash_T e_1 : \mathbf{f}}{\Gamma \vdash_T \text{cons}(e_1, e_2) : \mathbf{f}_\epsilon}$
		Case-1	$\frac{\Gamma \vdash_T e_3 : \mathbf{f}}{\Gamma \vdash_T \text{case}(e_1, e_2, e_3) : \mathbf{f}}$
		Case-2	$\frac{\Gamma \vdash_T e_2 : \mathbf{t} \rightarrow \infty \rightarrow \phi \quad \Gamma \vdash_T e_3 : \infty}{\Gamma \vdash_T \text{case}(e_1, e_2, e_3) : \phi}$
Case-3	$\frac{\Gamma \vdash_T e_2 : \mathbf{t} \rightarrow \mathbf{f}_\epsilon \rightarrow \phi \wedge \mathbf{f} \rightarrow \mathbf{t} \rightarrow \phi \quad \Gamma \vdash_T e_3 : \mathbf{f}_\epsilon}{\Gamma \vdash_T \text{case}(e_1, e_2, e_3) : \phi}$		
		Case-4	$\frac{\Gamma \vdash_T e_1 : \phi \quad \Gamma \vdash_T e_2 : \mathbf{t} \rightarrow \mathbf{t} \rightarrow \phi}{\Gamma \vdash_T \text{case}(e_1, e_2, e_3) : \phi}$
Taut-hd	$\Gamma \vdash_T \text{hd}(e) : \mathbf{t}$	Taut-tl	$\Gamma \vdash_T \text{tl}(e) : \mathbf{t}$
		Taut-cons	$\Gamma \vdash_T \text{cons}(e_1, e_2) : \mathbf{t}$

Fig. 2. The Strictness logic

We show in [13] that the most general type of an expression is precisely the information returned by the standard abstract interpretation-based analysis. This explains why abstract interpretation is sometimes inefficient because it may provide much more information than really required.

We take a different approach in this paper: rather than returning all possible information about the strictness of a function we compute only the information required to answer a particular question. This new philosophy naturally leads to a notion of lazy evaluation of types. The language of lazy types T_G is defined in Fig. 4. The ordering on types \leq_G and the logic \vdash_G are shown in Fig. 5.

The key idea is that an expression from the term language (with its environment) may appear as part of a type; this plays the rôle of a closure. More formally, a closure (Γ, e) stands for $MGT(\Gamma, e)$, the conjunction of all of the

$$\boxed{\begin{array}{c} t, f, \infty, f_\infty \in T_S \quad \frac{\sigma \in T_I \quad \psi \in T_S}{\sigma \rightarrow \psi \in T_S} \quad \frac{\phi_1 \in T_S \dots \phi_n \in T_S}{\phi_1 \wedge \dots \wedge \phi_n \in T_I} \end{array}}$$

Fig. 3. The language T_I

possible types of the term. This correspondence explains the new rules in the definition of \leq_G . Not surprisingly, the lazy evaluation of types is made explicit in the **App** rule: rather than deriving all possible types for e_2 , we insert e_2 itself (with the current environment) into the type of e_1 . The following definition establishes a correspondence between lazy types and ordinary types, the extension to environments is straightforward:

Definition 3.

$$\begin{aligned} \text{Expand} : T_G &\rightarrow T_I \\ \text{Expand}(t) &= t & \text{Expand}(f) &= f \\ \text{Expand}(\infty) &= \infty & \text{Expand}(f_\infty) &= f_\infty \\ \text{Expand}(\sigma_1 \wedge \sigma_2) &= \text{Expand}(\sigma_1) \wedge \text{Expand}(\sigma_2) \\ \text{Expand}(\sigma_1 \rightarrow \sigma_2) &= \text{Expand}(\sigma_1) \rightarrow \text{Expand}(\sigma_2) \\ \text{Expand}((\Gamma, e)) &= MGT(\text{Expand}(\Gamma), e) \end{aligned}$$

We can now state the correctness and completeness of the lazy type system and the subsequent equivalence with the original system.

Theorem 4 (Correctness).

$$\Gamma \vdash_G e : \phi \implies \text{Expand}(\Gamma) \vdash_T e : \text{Expand}(\phi) \quad \phi \in T_G$$

Theorem 5 (Completeness).

$$\text{Expand}(\Gamma) \vdash_T e : \text{Expand}(\phi) \implies \Gamma \vdash_G e : \phi \quad \phi \in T'_S$$

Theorem 6 (Equivalence).

$$\Gamma \vdash_T e : \phi \Leftrightarrow \Gamma \vdash_G e : \phi \quad \Gamma \in Var \rightarrow T_I, e : \phi \in T_I$$

First notice that we do not lose completeness by considering T_I types: it can be shown quite easily that any type is equivalent to a type in T_I . The following theorems are used in the proofs of theorems 4 and 5.

Theorem 7. $\sigma \leq_G \tau \Leftrightarrow \text{Expand}(\sigma) \leq \text{Expand}(\tau)$ **Theorem 8.**

$$\Gamma \vdash_G e : (\phi_1 \wedge \dots \wedge \phi_n) \Leftrightarrow (\Gamma \vdash_G e : \phi_1) \text{ and } \dots \text{ and } (\Gamma \vdash_G e : \phi_n)$$

$$\Gamma \vdash_T e : (\phi_1 \wedge \dots \wedge \phi_n) \Leftrightarrow (\Gamma \vdash_T e : \phi_1) \text{ and } \dots \text{ and } (\Gamma \vdash_T e : \phi_n)$$

Theorem 7 can be proved by induction on the proof of the left hand side. Theorem 8 is shown by deriving a proof of the right hand side from a proof of the left hand side (it is quite straightforward). Theorem 8 allows us to prove theorem 4 by induction on e . The proof of completeness is carried out in two stages. First we show that the weakening rule can be removed from \vdash_T without changing the set of derivable types provided we add a form of weakening in the **Var** and **Fix** rules. A similar property has been proved for other type systems including a form of weakening [1, 22]. Then we use theorems 7 and 8 and proceed by induction on e to prove completeness.

$\text{nil} \in \text{env}$	$\frac{\Gamma \in \text{env} \quad \sigma \in T_G}{\Gamma[x \mapsto \sigma] \in \text{env}}$	$\frac{\Gamma \in \text{env} \quad e \in \text{exp}}{(\Gamma, e) \in T_G}$
$t, f, \infty, f_e \in T'_S$	$\frac{\sigma \in T_G \quad \psi \in T'_S}{\sigma \rightarrow \psi \in T'_S}$	$\frac{\phi_1 \in T'_S \dots \phi_n \in T'_S}{\phi_1 \wedge \dots \wedge \phi_n \in T_G}$

Fig. 4. The language T_G

4 The lazy types algorithm

In [13], we show how to derive an abstract machine from the basic lazy types system. A similar derivation from the system defined in Section 3 leads to the machine shown in Fig 6. The state of the machine is a triple specifying the current contents of the stack, environment and code. $\text{Inf}(\phi, \psi)$ computes $\phi \leq_G \psi$ as defined in Fig. 5 (Theorem 10). Notice that a stack element S_i is either a boolean value or a disjunction of types. *True* (resp. *False*) is installed at the top of the stack if and only if the original property (of the form (e, ϕ)) in the code is (resp. is not) provable in \vdash_G . Values which are neither *True* nor *False* in the stack are disjunctions of T_G types ($\phi_1 \vee \dots \vee \phi_n$). The occurrence of such a value at the top of the stack means that the original property is true if (and only if) the recursive function currently being analysed possesses one of the ϕ_i types (in order to make the presentation simpler we do not consider embedded occurrences of **fix** here; the extension is straightforward). In order to prove that $\text{fix}(\lambda g.e)$ has type ϕ we add the assumption $(g :_r \phi)$ in the environment and try to prove $e : \phi$. If the result is *True* or *False* then the case is settled. Otherwise a disjunction of conditions ϕ_i is returned and the algorithm iterates to try to show that one of them is satisfied (rule for *Iter*). Instruction *Rec* is used to remember that we were trying to prove a property on a recursively defined variable (denoted by \mapsto_r in the environment); so if it fails we just return this property in the stack rather than *False*.

Primitives *And* and *Or* are extended in the obvious way to apply on types: their result is always supposed to be a disjunction of T_G types.

$\mathbf{f} \leq_G \phi \quad \phi \leq_G \phi \quad \infty \leq_G \mathbf{f}_\epsilon \quad \phi \leq_G t$ $\phi_1 \rightarrow \dots \rightarrow \phi_n \rightarrow \phi \leq_G \psi_1 \rightarrow \dots \rightarrow \psi_n \rightarrow t$ $\frac{\forall j \in [1, m], \exists i \in [1, n] \phi_i \leq_G \psi_j}{\phi_1 \wedge \dots \wedge \phi_n \leq_G \psi_1 \wedge \dots \wedge \psi_m} \quad \frac{\forall \phi. (\Gamma \vdash_G e : \phi) \Rightarrow \psi \leq_G \phi}{\psi \leq_G (\Gamma, e)}$ $\text{Conj} \quad \frac{\Gamma \vdash_G e : \psi_1 \quad \Gamma \vdash_G e : \psi_2}{\Gamma \vdash_G e : \psi_1 \wedge \psi_2} \quad \text{Var} \quad \frac{\psi_1 \leq_G \psi_2}{\Gamma[x \mapsto \psi_1] \vdash_G x : \psi_2}$ $\text{Abs} \quad \frac{\Gamma[x \mapsto \phi] \vdash_G e : \psi}{\Gamma \vdash_G \lambda x. e : (\phi \rightarrow \psi)} \quad \text{Taut} \quad \Gamma \vdash_G c : t$ $\text{App} \quad \frac{\Gamma \vdash_G e_1 : ((\Gamma, e_2) \rightarrow \psi)}{\Gamma \vdash_G e_1 e_2 : \psi}$ $\text{Fix} \quad \frac{\Gamma \vdash_G (\lambda g. e) : (\bigwedge_{i=1}^n \phi_i \rightarrow \phi_1) \wedge \dots \wedge (\bigwedge_{i=1}^n \phi_i \rightarrow \phi_n)}{\Gamma \vdash_G \text{fix}(\lambda g. e) : \phi_k \quad (k \in [1, n])}$ $\text{Cond-1} \quad \frac{\Gamma \vdash_G e_1 : \mathbf{f}}{\Gamma \vdash_T \text{cond}(e_1, e_2, e_3) : \phi} \quad \text{Cond-2} \quad \frac{\Gamma \vdash_G e_2 : \phi \quad \Gamma \vdash_G e_3 : \phi}{\Gamma \vdash_T \text{cond}(e_1, e_2, e_3) : \phi}$ $\text{Hd} \quad \frac{\Gamma \vdash_T e : \mathbf{f}}{\Gamma \vdash_T \text{hd}(e) : \phi} \quad \text{Tl-1} \quad \frac{\Gamma \vdash_T e : \mathbf{f}}{\Gamma \vdash_T \text{tl}(e) : \mathbf{f}} \quad \text{Tl-2} \quad \frac{\Gamma \vdash_T e : \infty}{\Gamma \vdash_T \text{tl}(e) : \infty}$ $\text{Tl-3} \quad \frac{\Gamma \vdash_T e : \infty}{\Gamma \vdash_T \text{tl}(e) : \mathbf{f}_\epsilon} \quad \text{Cons-1} \quad \frac{\Gamma \vdash_T e_2 : \infty}{\Gamma \vdash_T \text{cons}(e_1, e_2) : \infty}$ $\text{Cons-2} \quad \frac{\Gamma \vdash_T e_2 : \mathbf{f}_\epsilon}{\Gamma \vdash_T \text{cons}(e_1, e_2) : \mathbf{f}_\epsilon} \quad \text{Cons-3} \quad \frac{\Gamma \vdash_T e_1 : \mathbf{f}}{\Gamma \vdash_T \text{cons}(e_1, e_2) : \mathbf{f}_\epsilon}$ $\text{Case-1} \quad \frac{\Gamma \vdash_T e_3 : \mathbf{f}}{\Gamma \vdash_T \text{case}(e_1, e_2, e_3) : \phi}$ $\text{Case-2} \quad \frac{\Gamma \vdash_T e_2 : t \rightarrow \infty \rightarrow \phi \quad \Gamma \vdash_T e_3 : \infty}{\Gamma \vdash_T \text{case}(e_1, e_2, e_3) : \phi}$ $\text{Case-3} \quad \frac{\Gamma \vdash_T e_2 : t \rightarrow \mathbf{f}_\epsilon \rightarrow \phi \wedge \mathbf{f} \rightarrow t \rightarrow \phi \quad \Gamma \vdash_T e_3 : \mathbf{f}_\epsilon}{\Gamma \vdash_T \text{case}(e_1, e_2, e_3) : \phi}$ $\text{Case-4} \quad \frac{\Gamma \vdash_T e_1 : \phi \quad \Gamma \vdash_T e_2 : t \rightarrow t \rightarrow \phi}{\Gamma \vdash_T \text{case}(e_1, e_2, e_3) : \phi}$ $\text{Taut-hd} \quad \Gamma \vdash_T \text{hd}(e) : t \quad \text{Taut-tl} \quad \Gamma \vdash_T \text{tl}(e) : t$ $\text{Taut-cons} \quad \Gamma \vdash_T \text{cons}(e_1, e_2) : t$

Fig. 5. The Lazy Types system

$$\begin{aligned}
& \langle S, E, (c, t) : C \rangle \triangleright_G \langle \text{True} : S, E, C \rangle \\
& \langle S, E, (c, f) : C \rangle \triangleright_G \langle \text{False} : S, E, C \rangle \\
& \langle S, E, (e, \phi_1 \wedge \phi_2) : C \rangle \triangleright_G \langle S, E, (e, \phi_1) : (e, \phi_2) : \text{And} : C \rangle \\
& \langle S, E, (\lambda x. e, \sigma \rightarrow \tau) : C \rangle \triangleright_G \langle S, (x : \sigma) : E, (e, \tau) : D(x) : C \rangle \\
& \langle S, E, (e_1 e_2, \phi) : C \rangle \triangleright_G \langle S, E, (e_1, (E, e_2) \rightarrow \phi) : C \rangle \\
& \langle S, E[x \mapsto \phi], (x, \psi) : C \rangle \triangleright_G \langle S, E[x \mapsto \phi], \text{Inf}(\phi, \psi) : C \rangle \\
& \langle S, E, (\text{cond}(e_1, e_2, e_3), \phi) : C \rangle \triangleright_G \langle S, E, (e_1, f) : (e_2, \phi) : (e_3, \phi) : \text{And} : \text{Or} : C \rangle \\
& \langle S, (x : \sigma) : E, (D(x)) : C \rangle \triangleright_G \langle S, E, C \rangle \\
& \quad \langle S, E, (\text{fix}(\lambda g. e), \phi) : C \rangle \triangleright_G \langle S, (g :_r \phi) : E, (e, \phi) : \text{Iter}(g, e) : C \rangle \\
& \quad \langle S, E[g \mapsto_r \phi], (g, \psi) : C \rangle \triangleright_G \langle S, E[g \mapsto_r \phi], \text{Inf}(\phi, \psi) : (\text{Rec}, g, \psi) : C \rangle \\
& \langle \text{True} : S, E, (\text{Rec}, g, \phi) : C \rangle \triangleright_G \langle \text{True} : S, E, C \rangle \\
& \quad \langle S_1 : S, E, (\text{Rec}, g, \phi) : C \rangle \triangleright_G \langle \phi : S, E, C \rangle \\
& \quad S_1 \neq \text{True} \\
& \langle S_1 : S, (g :_r \phi) : E, \text{Iter}(g, e) : C \rangle \triangleright_G \langle S_1 : S, E, C \rangle \\
& \quad S_1 = \text{True} \quad \text{or} \quad S_1 = \text{False} \\
& \langle (\phi_1 \vee \dots \vee \phi_n) : S, (g :_r \phi) : E, \text{Iter}(g, e) : C \rangle \triangleright_G \\
& \quad \langle S, E, (\text{fix}(\lambda g. e), \phi_1) : (\text{fix}(\lambda g. e), \phi_2) : \text{Or} : \dots : \text{Or} : C \rangle \\
& \langle S, E, (\text{hd}(e), t) : C \rangle \triangleright_G \langle \text{True} : S, E, C \rangle \\
& \langle S, E, (\text{hd}(e), f) : C \rangle \triangleright_G \langle S, E, (e, f) : C \rangle \\
& \langle S, E, (\text{tl}(e), \infty) : C \rangle \triangleright_G \langle S, E, (e, \infty) : C \rangle \\
& \langle S, E, (\text{tl}(e), f_e) : C \rangle \triangleright_G \langle S, E, (e_1, f) : (e_2, f_e) : \text{Or} : C \rangle \\
& \langle S, E, (\text{cons}(e_1, e_2), f) : C \rangle \triangleright_G \langle \text{False} : S, E, C \rangle \\
& \langle S, E, (\text{case}(e_1, e_2, e_3), \phi) : C \rangle \triangleright_G \\
& \langle S, E, (e_3, f) : (e_2, t \rightarrow \infty \rightarrow \phi) : (e_3, \infty) : \text{And} : (e_2, t \rightarrow f_e \rightarrow \phi \wedge f \rightarrow t \rightarrow \phi) : (e_3, f_e) : \\
& \text{And} : (e_1, \phi) : (e_2, t \rightarrow t \rightarrow \phi) : \text{And} : \text{Or} : \text{Or} : C \rangle \\
& \langle S_1 : S_2 : S, E, \text{Op} : C \rangle \triangleright_G \langle (\text{Op } S_1 \text{ } S_2) : S, E, C \rangle \\
& \quad \text{Op} = \text{And} \quad \text{or} \quad \text{Op} = \text{Or}
\end{aligned}$$

Fig. 6. The Lazy Types algorithm

The following theorem states the correctness of the lazy types algorithm.

Theorem 9.

1. $\langle S, \Gamma, (e, \phi) : C \rangle \triangleright_G^* \langle \text{True} : S, \Gamma, C \rangle \Leftrightarrow \Gamma \vdash_G e : \phi$
2. $\langle S, \Gamma, (e, \phi) : C \rangle \triangleright_G^* \langle \text{False} : S, \Gamma, C \rangle \Leftrightarrow \neg(\Gamma \vdash_G e : \phi)$
if Γ and ϕ do not contain any \mapsto_r assumption

The proof of this theorem is made hand in hand with the proof of the following result:

Theorem 10.

1. $\langle S, \Gamma, \text{Inf}(\phi, \psi) : C \rangle \triangleright_G^* \langle \text{True} : S, \Gamma, C \rangle \Leftrightarrow \phi \leq_G \psi$
2. $\langle S, \Gamma, \text{Inf}(\phi, \psi) : C \rangle \triangleright_G^* \langle \text{False} : S, \Gamma, C \rangle \Leftrightarrow \neg(\phi \leq_G \psi)$
if Γ , ϕ and ψ do not contain any \mapsto_r assumption

The most difficult part of the proof concerns the implementation of **fix**. We have two main facts to prove: (1) the iteration terminates and (2) the result is accurate. Termination is proved by showing that each type $\phi \wedge \phi_i$ satisfies $\phi \wedge \phi_i <_G \phi$. It is easy to show that the result is accurate when the iteration terminates with the *True* answer. In order to show that the initial property cannot be satisfied if the answer is *False*, we prove that at least one of the ϕ_i types returned by the iteration step is a necessary condition to prove the original property (in other words, we do not “bypass” the least fixed point).

The algorithm described in this section can be optimised in several ways:

- The implementation of the conditional can avoid processing the second and third term when the first term has type **f**.
- In the same way, the implementation of the case operation can be considerably optimised if the first term has type **f**. More generally, *And* and *Or* can be modified in order to avoid the computation of their second argument when their first argument reduces respectively to *False* and *True*.
- In the rule for application, when expression e_2 is a constant or a variable then its type (**t** for a constant, its type in the environment for a variable) can be inserted into the type of e_1 rather than passing the whole environment. Notice that this optimisation is common in the implementation of lazy languages.

These optimisations are easy to justify formally and improve the derivation considerably.

5 Example

We consider the following functions:

$$\begin{aligned} \text{foldr } b \ g \ \text{nil} &= b \\ \text{foldr } b \ g \ \text{cons}(x, xs) &= g \ x \ (\text{foldr } b \ g \ xs) \\ \text{cat } l &= \text{foldr } \text{nil} \ \text{append } l \end{aligned}$$

which were introduced in [17] to demonstrate the inefficiency of traditional abstract interpretation. Notice that we have used pattern matching in the definition of *foldr*; this is for clarity - more properly it should have been defined as:

$$\text{foldr} = \text{fix}(\lambda f.\lambda b.\lambda g.\lambda l.\text{case}(b, \lambda x\lambda xs.g x (f b g xs), l))$$

Similarly *cat* should also be defined as a λ -abstraction.

Fig. 7 describes some of the derivation steps of the lazy type algorithm to prove that *cat* has type $\mathbf{f} \rightarrow \mathbf{f}$.

$$\begin{aligned}
 & \langle Nil, Nil, (\text{cat}, \mathbf{f} \rightarrow \mathbf{f}) \rangle && \triangleright_G \\
 & \langle Nil, (l : \mathbf{f}), (\text{foldr nil append } l, \mathbf{f}) : D(l) \rangle && \triangleright_G \\
 & \langle Nil, (l : \mathbf{f}), (\text{foldr nil append}, \mathbf{f} \rightarrow \mathbf{f}) : D(l) \rangle && \triangleright_G \\
 & \langle Nil, (l : \mathbf{f}), (\text{foldr nil}, ((l : \mathbf{f}), \text{append}) \rightarrow \mathbf{f} \rightarrow \mathbf{f}) : D(l) \rangle && \triangleright_G \\
 & \langle Nil, (l : \mathbf{f}), (\text{foldr}, t \rightarrow ((l : \mathbf{f}), \text{append}) \rightarrow \mathbf{f} \rightarrow \mathbf{f}) : D(l) \rangle && \triangleright_G \\
 & \vdots && \\
 & \langle Nil, (l : \mathbf{f}) : (g, ((l : \mathbf{f}), \text{append})) : (b : t) : (f :_r \phi) : (l : \mathbf{f}), \\
 & \quad (\text{case } \dots, \mathbf{f}) : D(l) : D(g) : D(b) : \text{Iter}(f, \dots) : D(l) \rangle && \triangleright_G \\
 & \vdots && \\
 & \langle Nil, \dots, (l, \mathbf{f}) : \dots : \text{Or} : D(l) : D(g) : D(b) : \dots \rangle && \triangleright_G \\
 & \vdots && \\
 & \langle True, \dots, D(l) : D(g) : D(b) : \dots \rangle && \triangleright_G \\
 & \vdots && \\
 & \langle True, (f :_r \phi) : (l : \mathbf{f}) : \dots, \text{Iter}(f, \dots) : D(l) \rangle && \triangleright_G \\
 & \langle True, (l : \mathbf{f}), D(l) \rangle && \triangleright_G \\
 & \langle True, Nil, Nil \rangle &&
 \end{aligned}$$

where ϕ is $t \rightarrow ((l : \mathbf{f}), \text{append}) \rightarrow \mathbf{f} \rightarrow \mathbf{f}$.

Fig. 7. *cat* has type $\mathbf{f} \rightarrow \mathbf{f}$

6 Generalisation to domains of any depth

The 4-point domain expresses information about lists with atomic elements. For example, it is not adequate for describing a property such as “this is a list containing lists whose one element is undefined”. Following Wadler [25], we can in fact generalise the definition of 4-point domain from the 2-point domain to domains of any depth. Let

$$D_0 = \{\mathbf{t}, \mathbf{f}\}$$

with $\mathbf{f} \leq_0 \mathbf{t}$. Then

$$D_{i+1} = \{\mathbf{f}, \infty\} \cup \{x_\in \mid x \in D_i\}$$

with:

$$\mathbf{f} \leq_{i+1} \infty$$

$$\forall x_\epsilon \in D_{i+1}. \infty \leq_{i+1} x_\epsilon$$

$$\forall x_\epsilon, y_\epsilon \in D_{i+1}. x \leq_i y \Leftrightarrow x_\epsilon \leq_{i+1} y_\epsilon$$

The following property shows that we can omit the subscript and write \leq for \leq_i :

$$\forall x, y \in D_i \cap D_{i+1}. x \leq_i y \Leftrightarrow x \leq_{i+1} y$$

An interesting property of our type inference system (and algorithm) is that it can be generalised without further complication to domains of unbounded depth. The rules **Cons-2**, **Cons-3** and **Case-3** are generalised in the following way:

$$\begin{array}{c} \text{Cons-2} \quad \frac{\Gamma \vdash_T e_2 : \sigma_\epsilon}{\Gamma \vdash_T \mathbf{cons}(e_1, e_2) : \sigma_\epsilon} \quad \text{Cons-3} \quad \frac{\Gamma \vdash_T e_1 : \sigma}{\Gamma \vdash_T \mathbf{cons}(e_1, e_2) : \sigma_\epsilon} \\ \\ \text{Case-3} \quad \frac{\Gamma \vdash_T e_2 : t \rightarrow \sigma_\epsilon \rightarrow \phi \wedge \sigma \rightarrow t \rightarrow \phi \quad \Gamma \vdash_T e_3 : \sigma_\epsilon}{\Gamma \vdash_T \mathbf{case}(e_1, e_2, e_3) : \phi} \end{array}$$

and the ordering on types is extended with the rules:

$$\infty \leq \sigma_\epsilon \quad \frac{\sigma \leq \tau}{\sigma_\epsilon \leq \tau_\epsilon}$$

The extensions to the algorithm are not described here for the sake of brevity. The implementation of **Cons-2** and **Cons-3** is straightforward because all the free variables occurring in the premises appear in the conclusion. This is not the case for **Case-3** which requires an iteration very much like the rule for abstraction in Fig. 6. The iteration explores the domain starting with D_0 until the property is proven or the maximal depth corresponding to the type of the expression is reached. Several trivial optimisations can dramatically improve the algorithm at this stage. For instance e_3 will often be a variable whose type is defined in the environment (see example below) and can be used to make the appropriate choice of σ , thus avoiding the iteration mentioned above.

We continue the *foldr* example to show that our system (and algorithm) does not need a domain of fixed depth but rather explores the potentially infinite domain up to the depth required to answer a particular question. We first restate the definition of *append* as a term of Λ_L :

$$\mathit{append} = \mathbf{fix}(\lambda \mathit{app}. \lambda x_1. \lambda x_2. \mathbf{case}(x_2, \lambda x. \lambda xs. \mathbf{cons}(x, (\mathit{app} \; xs \; x_2)), x_1))$$

Assume that we want to prove $\mathit{foldr} : t \rightarrow \mathit{append} \rightarrow \infty_\epsilon \rightarrow \infty$, where append is used as a shorthand notation for $(\emptyset, \mathit{append})$. We do not give all of

the details of the derivation but rather focus on the main steps of the proof:

$$\begin{array}{c}
 \vdots \\
 \text{Conj} \quad \frac{}{\Gamma \vdash (\lambda x. \lambda xs. g\ x\ (f\ b\ g\ xs)) : (t \rightarrow \infty_\epsilon \rightarrow \infty) \wedge (\infty \rightarrow t \rightarrow \infty)} \quad C \\
 \text{Case - 3} \quad \frac{}{\Gamma \vdash \text{case}(b, \lambda x. \lambda xs. g\ x\ (f\ b\ g\ xs), l) : \infty} \\
 \vdots \\
 \text{Abs} \quad \frac{}{\vdash \lambda f. \lambda b. \lambda g. \lambda l. \text{case}(b, \lambda x. \lambda xs. g\ x\ (f\ b\ g\ xs), l) : \\
 \quad (t \rightarrow \text{append} \rightarrow \infty_\epsilon \rightarrow \infty) \rightarrow (t \rightarrow \text{append} \rightarrow \infty_\epsilon \rightarrow \infty)} \\
 \text{Fix} \quad \frac{\vdash \text{fix}(\lambda f. \lambda b. \lambda g. \lambda l. \text{case}(b, \lambda x. \lambda xs. g\ x\ (f\ b\ g\ xs), l)) : t \rightarrow \text{append} \rightarrow \infty_\epsilon \rightarrow \infty}{\vdash \text{foldr} : t \rightarrow \text{append} \rightarrow \infty_\epsilon \rightarrow \infty} \\
 \vdots \\
 \text{where } \Gamma \text{ is: } [f : t \rightarrow \text{append} \rightarrow \infty_\epsilon \rightarrow \infty, b : t, g : \text{append}, l : \infty_\epsilon]. \text{ A is:} \\
 \vdots \\
 \frac{\Gamma'' \vdash f\ b\ g\ xs : \infty}{\vdash (\Gamma'', f\ b\ g\ xs) \leq \infty} \\
 \vdots \\
 \text{Case - 4} \quad \frac{\Gamma' \vdash \lambda x. \lambda xs. \text{cons}(x, (\text{app}\ xs\ x_2)) : t \rightarrow t \rightarrow \infty \quad \frac{}{\Gamma' \vdash x_2 : \infty}}{\Gamma' \vdash \text{case}(x_2, \lambda x. \lambda xs. \text{cons}(x, (\text{app}\ xs\ x_2)), x_1) : \infty} \\
 \vdots \\
 \text{App} \quad \frac{\Gamma'' \vdash g : t \rightarrow (f\ b\ g\ xs) \rightarrow \infty}{\vdash} \\
 \text{App} \quad \frac{\Gamma'' \vdash g\ x : (f\ b\ g\ xs) \rightarrow \infty}{\vdash} \\
 \text{App} \quad \frac{\Gamma'' \vdash g\ x : (f\ b\ g\ xs) : \infty}{\vdash} \\
 \vdots \\
 \text{Abs} \quad \frac{\Gamma \vdash (\lambda x. \lambda xs. g\ x\ (f\ b\ g\ xs)) : (t \rightarrow \infty_\epsilon \rightarrow \infty)}{\vdash}
 \end{array}$$

where

$$\begin{aligned}
 \Gamma' &= [\text{app} : (t \rightarrow (\Gamma'', (f\ b\ g\ xs)) \rightarrow \infty), x_1 : t, x_2 : (\Gamma'', (f\ b\ g\ xs))] : \Gamma'' \\
 \Gamma'' &= [x : t, xs : \infty_\epsilon] : \Gamma
 \end{aligned}$$

the proof tree for B is similarly constructed and C is $\Gamma \vdash l : \infty_\epsilon$. So the domain is explored up to depth 2 (D_2). If we now ask the question $\text{foldr} : t \rightarrow \text{append} \rightarrow \mathbf{f}_\epsilon \rightarrow \infty$, the domain is not explored further than depth 1, as the reader can easily verify (the structure of the proof is very similar to the previous one).

7 Conclusions

The problem of designing efficient algorithms for strictness analysis has received much attention recently and one current trend seems to revert from the usual “extensional” approach to more “intensional” or syntactic techniques [20, 21, 18, 6, 10, 24]. The key observation underlying these works is that the choice of representing abstract functions by functions can be disastrous in terms of efficiency and is not always justified in terms of accuracy. Some of these proposals trade a cheaper implementation against a loss of accuracy [20, 21]. In contrast, [10, 24] use extensional representations of functions to build very efficient algorithms without sacrificing accuracy. The analysis of [10] uses concrete

data structures; these are special kinds of Scott domains whose elements can be seen as syntax trees. In [24] the analysis is expressed as a form of reduction of abstract graphs. An interesting avenue for further research would be to reexpress these analyses in terms of type inference as suggested here to prove their correctness and to be able to relate the techniques on a formal basis.

Wadler's domain construction does not readily generalise to other recursive data types. Recently Benton [3] has shown how to construct an abstract domain from any algebraic data type. It should be straightforward to extend our system (and algorithm) to incorporate such domains. Benton's construction leads to quite large domains; the size of the domains would make conventional abstract interpretation intractable and highlights the benefit of our approach which lazily explores the domain.

In his thesis Jensen, [19], has developed a more general logical treatment of recursive types. His approach involves two extensions to the logic; the first is to add disjunctions and the second extension involves adding modal operators for describing uniform properties of elements of recursive types. The extension of our techniques to these richer logics is an open research problem which we are currently investigating.

References

1. S. van Bakel, *Complete restrictions of the intersection type discipline*, Theoretical Computer Science, 102(1):135-163, 1992.
2. P. N. Benton, *Strictness logic and polymorphic invariance*, in *Proceedings of the 2nd Int. Symposium on Logical Foundations of Computer Science*, LNCS 620, Springer Verlag, 1992.
3. P. N. Benton, *Strictness Properties of Lazy Algebraic Datatypes*, in *Proceedings WSA'93*, LNCS 724, Springer Verlag, 1993.
4. G. L. Burn, *Evaluation Transformers - a model for the parallel evaluation of functional languages (extended abstract)*, in *Proceedings of the 1987 Conference on Functional Programming Languages and Computer Architecture*, LNCS 274, Springer Verlag, 1987.
5. G. Burn and D. Le Métayer, *Proving the correctness of compiler optimisations based on strictness analysis*, in *Proceedings 5th int. Symp. on Programming Language Implementation and Logic Programming*, LNCS 714, Springer Verlag, 1993.
6. T.-R. Chuang and B. Goldberg, *A syntactic approach to fixed point computation on finite domains*, in *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, ACM Press, 1992.
7. P. Cousot and R. Cousot, *Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation*, in M. Bruynooghe and M. Wirsing (eds), *PLILP'92*, LNCS 631, Springer Verlag, 1992.
8. O. Danvy and J. Hatcliff, *CPS transformation after strictness analysis*, Technical Report, Kansas State University, to appear in ACM LOPLAS.
9. M. van Eekelen, E. Goubault, C. Hankin and E. Nøker, *Abstract reduction: a theory via abstract interpretation*, in R. Sleep et al (eds), *Term graph rewriting: theory and practice*, John Wiley & Sons Ltd, 1992.
10. A. Ferguson and R. J. M. Hughes, *Fast abstract interpretation using sequential algorithms*, in *Proceedings WSA'93*, LNCS 724, Springer Verlag, 1993.

11. S. Finne and G. Burn, *Assessing the evaluation transformer model of reduction on the spineless G-machine*, in *Proceedings of the 6th ACM Conference on Functional Programming Languages and Computer Architecture*, ACM Press, 1993, pp. 331-341.
12. C. L. Hankin and L. S. Hunt, *Approximate fixed points in abstract interpretation*, in B. Krieg-Brückner (ed), *Proceedings of the 4th European Symposium on Programming*, LNCS 582, Springer Verlag, 1992.
13. C. L. Hankin and D. Le Métayer, *Deriving algorithms from type inference systems: Application to strictness analysis*, to appear in *Proceedings of POPL'94*, ACM Press, 1994.
14. J. J. Hannan, *Investigating a proof-theoretic meta-language*, PhD thesis, University of Pennsylvania, DIKU Technical Report Nr 91/1, 1991.
15. J. Hannan and D. Miller, *From Operational Semantics to Abstract Machines*, Mathematical Structures in Computer Science, 2(4), 1992.
16. P. H. Hartel and K. G. Langendoen, *Benchmarking implementations of lazy functional languages*, in *Proceedings of the 6th ACM Conference on Functional Programming Languages and Computer Architecture*, ACM Press, 1993, pp. 341-350.
17. L. S. Hunt and C. L. Hankin, *Fixed Points and Frontiers: A New Perspective*, Journal of Functional Programming, 1(1), 1991.
18. T. P. Jensen, *Strictness Analysis in Logical Form*, in J. Hughes (ed), *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, LNCS 523, Springer Verlag, 1991.
19. T. P. Jensen, *Abstract Interpretation in Logical Form*, PhD thesis, University of London, 1992. Also available as DIKU Technical Report 93/11.
20. T.-M. Kuo and P. Mishra, *Strictness analysis: a new perspective based on type inference*, in *Proceedings of the 4th ACM Conference on Functional Programming Languages and Computer Architecture*, ACM Press, 1989.
21. A. Leung and P. Mishra, *Reasoning about simple and exhaustive demand in higher-order lazy languages*, in *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, LNCS 523, Springer Verlag, 1991.
22. J. C. Mitchell, *Type inference with simple subtypes*, Journal of Functional Programming, 1(3), 1991.
23. A. Mycroft, *Abstract Interpretation and Optimising Transformations for Applicative Programs*, PhD thesis, University of Edinburgh, December 1981.
24. E. Nöcker, *Strictness analysis using abstract reduction*, in *Proceedings of the 6th ACM Conference on Functional Programming Languages and Computer Architecture*, ACM Press, 1993.
25. P. Wadler, *Strictness Analysis on Non-flat Domains*, in S. Abramsky and C. L. Hankin (eds), *Abstract Interpretation of Declarative Languages*, Ellis Horwood, 1987.
26. P. Wadler and J. Hughes, *Projections for Strictness Analysis*, in *Proceedings of the 1987 Conference on Functional Programming Languages and Computer Architecture*, LNCS 274, Springer Verlag, 1987.

Lazy Unification with Simplification

Michael Hanus

Max-Planck-Institut für Informatik
Im Stadtwald, D-66123 Saarbrücken, Germany.
michael@mpi-sb.mpg.de

Abstract. Unification in the presence of an equational theory is an important problem in theorem-proving and in the integration of functional and logic programming languages. This paper presents an improvement of the proposed lazy unification methods by incorporating simplification into the unification process. Since simplification is a deterministic computation process, more efficient unification algorithms can be achieved. Moreover, simplification reduces the search space so that in some case infinite search spaces are reduced to finite ones. We show soundness and completeness of our method for equational theories represented by ground confluent and terminating rewrite systems which is a reasonable class w.r.t. functional logic programming.

1 Introduction

Unification is not only an important operation in theorem provers but also the most important operation in logic programming systems. Unification in the presence of an equational theory, also known as *E-unification*, is necessary if the computational domain in a theorem prover enjoys certain equational properties [26] or if functions should be integrated into a logic language [10]. Therefore the development of E-unification algorithms is an active research topic during recent years (see, for instance, [29]).

Since E-unification is a complex problem even for simple equational axioms, we are interested in efficient E-unification methods in order to incorporate such methods into functional logic programming languages. One general method to improve the efficiency of implementations is the use of a *lazy strategy*. “Lazy” means that evaluations are performed only if it is necessary to compute the required solutions. In the context of unification this corresponds to the idea that terms are manipulated at outermost positions. Hence *lazy unification* means that equational axioms are applied to outermost positions of equations. For instance, consider the following equations for addition and multiplication on natural numbers which are represented by terms of the form $s(\dots s(0) \dots)$:

$$\begin{array}{ll} 0 + y \approx y & 0 * y \approx 0 \\ s(x) + y \approx s(x + y) & s(x) * y \approx y + x * y \end{array}$$

If we have to unify the terms $0 * (s(0) + s(z))$ and 0 , we could apply equational axioms to inner subterms starting with $s(0) + s(z)$ (*innermost* or *eager strategy*) or to outermost subterms (*outermost* or *lazy strategy*). This will lead to the following two derivations (the subterms manipulated in the next step are underlined):

$$\begin{aligned} 0 * (s(0) + s(z)) \approx 0 &\Rightarrow 0 * (\underline{s(0 + s(z))}) \approx 0 \Rightarrow \underline{0 * (s(s(z)))} \approx 0 \Rightarrow 0 \approx 0 \\ \underline{0 * (s(0) + s(z))} \approx 0 &\Rightarrow 0 \approx 0 \end{aligned}$$

Obviously, the second lazy unification derivation should be preferred.

There are many proposals for such lazy unification strategies. For instance, Martelli et al. [22] have proposed a lazy unification algorithm for confluent and terminating equational axioms. Due to the confluence requirement, equations are only applied in one direction. However, their method is not pure lazy since equations are applied to inner subterms in equations of the form $x \approx t$ where the variable x occurs in t . Gallier and Snyder [11] have proved the completeness of a lazy unification method for arbitrary equational theories where equations can be applied in both directions. *Narrowing* is a method to compute E-unifiers in the presence of confluent axioms. It is a combination of the reduction principle of functional languages with syntactic unification in order to instantiate variables. Lazy narrowing were proposed by Reddy [27] as the operational principle of functional logic languages. Recently, Antoy, Echahed and Hanus [1] have proposed a narrowing strategy for programs where the functions are defined by case distinctions over the data structures. This strategy reduces only needed redexes, computes no redundant solutions, and is optimal w.r.t. the length of narrowing derivations.

From a practical point of view the disadvantage of E-unification is its inherent nondeterminism. In the area of narrowing there are many proposals for the inclusion of a deterministic simplification process in order to reduce the nondeterminism [8, 9, 19, 24, 28], but all these proposals are based on an eager narrowing strategy. On the other hand, only little work has been done to improve the efficiency of outermost or lazy strategies. Echahed [7] has shown the completeness of any narrowing strategy with simplification under strong requirements (uniformity of specifications). Dershowitz et al. [6] have proposed to combine lazy unification with simplification and demonstrated the usefulness of inductive consequences for simplification. However, they have not proved completeness of their lazy unification calculus if all terms are simplified to their normal form after each unification step. In fact, their completeness proof for lazy narrowing does not hold if eager rewriting is included since rewriting in their sense does not reduce the complexity measure used in their completeness proof and may lead to infinite instead of successful derivations. Therefore we will formulate a calculus for lazy unification which includes simplification and give a rigorous completeness proof. The distinguishing features of our framework are:

- We consider a ground confluent and terminating equational specification in order to apply equations only in one direction and to ensure the existence of normal forms. This is reasonable if one is interested in declarative programming rather than theorem proving.
- The unification calculus is lazy, i.e., functions are not evaluated if their value is not required to decide the unifiability of terms. Consequently, we may compute *reducible solutions* as answers according to the spirit of lazy evaluation. For instance, in contrast to other “lazy” unification methods we do not allow any evaluation of t in the equation $x \approx t$ if x occurs only once.
- We include a deterministic simplification process in our unification calculus. In order to restrict nondeterministic computations as much as possible, we allow to use additional inductive consequences for simplification which has been proved to be useful in other calculi [7, 9, 24].

After recalling basic notions from term rewriting, we present in Section 3 our basic lazy unification calculus. In Section 4 we include a deterministic simplification process into the lazy unification calculus. Finally, we show in Section 5 some important optimizations for constructor-based specifications. Due to lack of space we omit the details of some proofs, but the interested reader will find them in [17].

2 Computing in equational theories

In this section we recall the notations for equations and term rewriting systems [5] which are necessary in our context.

Let the *signature* \mathcal{F} be a set of *function symbols*¹ and \mathcal{X} be a countably infinite set of *variables*. Then $\mathcal{T}(\mathcal{F}, \mathcal{X})$ denotes the set of *terms* built from \mathcal{F} and \mathcal{X} . $\text{Var}(t)$ is the set of variables occurring in t . A *ground term* t is a term without variables, i.e., $\text{Var}(t) = \emptyset$. A *substitution* σ is a mapping from \mathcal{X} into $\mathcal{T}(\mathcal{F}, \mathcal{X})$ such that its *domain* $\text{Dom}(\sigma) = \{x \in \mathcal{X} \mid \sigma(x) \neq x\}$ is finite. We frequently identify a substitution σ with the set $\{x \mapsto \sigma(x) \mid x \in \text{Dom}(\sigma)\}$. Substitutions are extended to morphisms on $\mathcal{T}(\Sigma, \mathcal{X})$ by $\sigma(f(t_1, \dots, t_n)) = f(\sigma(t_1), \dots, \sigma(t_n))$ for every term $f(t_1, \dots, t_n)$. A substitution σ is called *ground* if $\sigma(x)$ is a ground term for all $x \in \text{Dom}(\sigma)$. The *composition of two substitutions* ϕ and σ is defined by $\phi \circ \sigma(x) = \phi(\sigma(x))$ for all $x \in \mathcal{X}$. A *unifier* of two terms s and t is a substitution σ with $\sigma(s) = \sigma(t)$. A unifier σ is called *most general (mgu)* if for every other unifier σ' there is a substitution ϕ with $\sigma' = \phi \circ \sigma$. A *position* p in a term t is represented by a sequence of natural numbers, $t|_p$ denotes the *subterm* of t at position p , and $t[s]_p$ denotes the result of *replacing the subterm* $t|_p$ by the term s (see [5] for details). The outermost position Λ is also called *root* position.

- Let \rightarrow be a binary relation on a set S . Then \rightarrow^* denotes the transitive and reflexive closure of the relation \rightarrow , and \leftrightarrow^* denotes the transitive, reflexive and symmetric closure of \rightarrow . \rightarrow is called *terminating* if there are no infinite chains $e_1 \rightarrow e_2 \rightarrow e_3 \rightarrow \dots$. \rightarrow is called *confluent* if for all $e, e_1, e_2 \in S$ with $e \rightarrow^* e_1$ and $e \rightarrow^* e_2$ there exists an element $e_3 \in S$ with $e_1 \rightarrow^* e_3$ and $e_2 \rightarrow^* e_3$.

An *equation* $s \approx t$ is a multiset containing two terms s and t . Thus equations to be unified are symmetric. In order to compute with equational specifications, we will use the specified equations only in one direction. Hence we define a *rewrite rule* $l \rightarrow r$ as a pair of terms l, r satisfying $l \notin \mathcal{X}$ and $\text{Var}(r) \subseteq \text{Var}(l)$ where l and r are called left-hand side and right-hand side, respectively. A rewrite rule is called a *variant* of another rule if it is obtained by a unique replacement of variables by other variables. A *term rewriting system* \mathcal{R} is a set of rewrite rules. In the following we assume a given *term rewriting system* \mathcal{R} .

A *rewrite step* is an application of a rewrite rule to a term, i.e., $t \rightarrow_{\mathcal{R}} s$ if there exists a position p , a rewrite rule $l \rightarrow r$ and a substitution σ with $t|_p = \sigma(l)$ and $s = t[\sigma(r)]_p$. A term t is called *reducible* if we can apply a rewrite rule to it, and t is called *irreducible* or in *normal form* if there is no term s with $t \rightarrow_{\mathcal{R}} s$. A term rewriting system is *ground confluent* if the restriction of $\rightarrow_{\mathcal{R}}$ to the set of all ground terms is confluent. If \mathcal{R} is ground confluent and terminating, then each ground term t has a unique normal form which is denoted by $t \downarrow_{\mathcal{R}}$.

We are interested in proving the validity of equations. Hence we call an equation $s \approx t$ *valid* (w.r.t. \mathcal{R}) if $s \leftrightarrow_{\mathcal{R}}^* t$. By Birkhoff's Completeness Theorem, this is equivalent to the validity of $s \approx t$ in all models of \mathcal{R} . In this case we also write $s =_{\mathcal{R}} t$. If \mathcal{R} is ground confluent and terminating, we can decide the validity of a ground equation $s \approx t$ by computing the normal form of both sides using an arbitrary sequence of rewrite steps since $s \leftrightarrow_{\mathcal{R}}^* t$ iff $s \downarrow_{\mathcal{R}} = t \downarrow_{\mathcal{R}}$. In order to compute *solutions* to a non-ground equation $s \approx t$, we have to find appropriate instantiations for the variables in s and t . This can be done by *narrowing*. A term

¹ In this paper we consider only single-sorted programs. The extension to many-sorted signatures is straightforward [25]. Since sorts are not relevant to the subject of this paper, we omit them for the sake of simplicity.

t is *narrowable* into a term t' if there exist a non-variable position p (i.e., $t|_p \notin \mathcal{X}$), a variant $l \rightarrow r$ of a rewrite rule and a substitution σ such that σ is a mgu of $t|_p$ and l and $t' = \sigma(t[r]_p)$. In this case we write $t \rightsquigarrow_{\sigma} t'$.

Narrowing is able to solve equations w.r.t. \mathcal{R} by deriving both sides of an equation to syntactically unifiable terms. Due to the huge search space of simple narrowing, several authors have proposed restrictions on the admissible narrowing derivations (see [18] for a detailed survey). *Lazy narrowing* [3, 23, 27] is influenced by the idea of lazy evaluation in functional programming languages. Lazy narrowing steps are only applied at outermost positions with the exception that arguments are evaluated by narrowing to their head normal form if their values are required for an outermost narrowing step. Since lazy strategies are important in the context of non-terminating rewrite rules, these strategies have been proved to be complete w.r.t. domain-based interpretations of rewrite rules [13, 23]. *Lazy unification* is very similar to lazy narrowing but manipulates sets of equations rather than terms. It has been proved to be complete for canonical term rewriting systems w.r.t. standard semantics [6, 22].

From a practical point of view the most essential improvement of simple narrowing is *normalizing narrowing* [8] where the term is rewritten to its normal form before a narrowing step is applied. This optimization is important since it prefers deterministic computations: rewriting a term to normal form can be done in a deterministic way since every rewriting sequence gives the same result (if \mathcal{R} is confluent and terminating). As shown in [9, 16], normalizing narrowing has the important effect that equational logic programs are more efficiently executable than pure logic programs. Normalization can also be combined with other narrowing restrictions [9, 19, 28]. Because of these important advantages, normalizing narrowing is the foundation of several programming languages which combines functional and logic programming like ALF [15], LPG [2] or SLOG [9]. However, normalization has not been included in lazy narrowing strategies.² Therefore we will present a lazy unification calculus which includes a normalization process where the term rewrite rules as well as additional inductive consequences are used for normalization.

3 A calculus for lazy unification

In the rest of this paper we assume that \mathcal{R} is a ground confluent and terminating term rewriting system. This section presents our basic lazy unification calculus to solve a system of equations. The inclusion of a normalization process will be shown in Section 4. The “laziness” of our calculus is in the spirit of lazy evaluation in functional programming languages, i.e., terms are evaluated only if their values are needed.

Our lazy unification calculus manipulates sets of equations in the style of Martelli and Montanari [21] rather than terms as in narrowing calculi. Hence we define an *equation system* E to be a multiset of equations (in the following we write such sets without curly brackets if it is clear from the context). A *solution* of an equation system E is a ground substitution σ such that $\sigma(s) =_{\mathcal{R}} \sigma(t)$ for all equations $s \approx t \in E$.³ An equation system E is *solvable* if it has at least one solution. A set S of substitutions is a *complete set of solutions* for E iff

² Except for [6, 7], but see the remarks in Section 1.

³ We are interested in *ground* solutions since later we will include inductive consequences which are valid in the ground models of \mathcal{R} . As pointed out in [24], this ground approach subsumes the conventional narrowing approaches where also non-ground solutions are taken into account.

Lazy narrowing

$$f(t_1, \dots, t_n) \approx t, E \xrightarrow{lu} t_1 \approx l_1, \dots, t_n \approx l_n, r \approx t, E$$

if $t \notin \mathcal{X}$ or $t \in \text{Var}(f(t_1, \dots, t_n)) \cup \text{Var}(E)$ and $f(l_1, \dots, l_n) \rightarrow r$ new variant of a rule

Decomposition of equations

$$f(t_1, \dots, t_n) \approx f(t'_1, \dots, t'_n), E \xrightarrow{lu} t_1 \approx t'_1, \dots, t_n \approx t'_n, E$$

Partial binding of variables

$$x \approx f(t_1, \dots, t_n), E \xrightarrow{lu} x \approx f(x_1, \dots, x_n), x_1 \approx \phi(t_1), \dots, x_n \approx \phi(t_n), \phi(E)$$

if $x \in \text{Var}(f(t_1, \dots, t_n)) \cup \text{Var}(E)$ and $\phi = \{x \mapsto f(x_1, \dots, x_n)\}$ (x_i new variable)

Figure 1. The lazy unification calculus

1. for all $\sigma \in S$, σ is a solution of E ;
2. for every solution θ of E , there exists some $\sigma \in S$ with $\theta(x) =_{\mathcal{R}} \sigma(x)$ for all $x \in \text{Var}(E)$.

In order to compute solutions of an equation system, we transform it by the rules in Figure 1 until no more rules can be applied. The lazy narrowing transformation applies a rewrite rule to a function occurring outermost in an equation.⁴ Actually, this is not a narrowing step as defined in Section 2 since the argument terms may not be unifiable. Narrowing steps can be simulated by a sequence of transformations in the lazy unification calculus but not vice versa since our calculus also allows the application of rewrite rules to the arguments of the left-hand sides. The decomposition transformation generates equations between the argument terms of an equation if both sides have the same outermost symbol. The partial binding of variables can be applied if the variable x occurs at different positions in the equation system. In this case we instantiate the variable only with the outermost function symbol. A full instantiation by the substitution $\phi = \{x \mapsto f(t_1, \dots, t_n)\}$ may increase the computational work if x occurs several times and the evaluation of $f(t_1, \dots, t_n)$ is costly. In order to avoid this problem of *eager variable elimination* (see [11]), we perform only a partial binding which is also called “root imitation” in [11].

At first sight our lazy unification calculus has many similarities with the lazy unification rules presented in [6, 11, 22, 25]. This is not accidental since these systems have inspired us. However, there are also essential differences. Since we are interested in reducing the computational costs in the E-unification procedure, our rules behave “more lazily”. In our rules it is allowed to evaluate a term only if its value is needed (in several positions). Otherwise, the term is left unevaluated.

Example 1. Consider the rewrite rule $0 * x \rightarrow 0$. Then the only transformation sequence of the equation $0 * t \approx 0$ (where t is a costly function) is

$$0 * t \approx 0 \xrightarrow{lu} 0 \approx 0, t \approx x, 0 \approx 0 \xrightarrow{lu} t \approx x, 0 \approx 0 \xrightarrow{lu} t \approx x$$

Thus the term t is not evaluated since its concrete value is not needed. Consequently, we may compute solutions with reducible terms which is a desirable property in the presence of a lazy evaluation mechanism. \square

⁴ Similarly to logic programming, we have to apply rewrite rules with fresh variables in order to ensure completeness.

Coalesce	$x \approx y, E \xrightarrow{\text{var}} x \approx y, \phi(E)$	if $x, y \in \text{Var}(E)$ and $\phi = \{x \mapsto y\}$
Trivial	$x \approx x, E \xrightarrow{\text{var}} E$	

Figure 2. The variable elimination rules

The conventional transformation rules for unification w.r.t. an empty equational theory [21] bind a variable x to a term t only if x does not occur in t . This *occur check* must be omitted in the presence of evaluable function symbols. Moreover, we must also instantiate occurrences of x in the term t which is done in our partial binding rule. The following example shows the necessity of these extensions.

Example 2. Consider the rewrite rule $f(c(a)) \rightarrow a$. Then we can solve the equation $x \approx c(f(x))$ by the following transformation sequence:

$$\begin{aligned} x \approx c(f(x)) &\xrightarrow{\text{lu}} x \approx c(x_1), x_1 \approx f(c(x_1)) && (\text{partial binding}) \\ &\xrightarrow{\text{lu}} x \approx c(x_1), c(x_1) \approx c(a), x_1 \approx a && (\text{lazy narrowing}) \\ &\xrightarrow{\text{lu}} x \approx c(x_1), x_1 \approx a, x_1 \approx a && (\text{decomposition}) \\ &\xrightarrow{\text{lu}} x \approx c(a), x_1 \approx a, a \approx a && (\text{partial binding}) \\ &\xrightarrow{\text{lu}} x \approx c(a), x_1 \approx a && (\text{decomposition}) \end{aligned}$$

In fact, the initial equation is solvable and $\{x \mapsto c(a)\}$ is a solution of this equation. This solution is also an obvious solution of the final equation system if we disregard the auxiliary variable x_1 . \square

In the rest of this section we will show soundness and completeness of our lazy unification calculus. Soundness simply means that each solution of the transformed equation system is also a solution of the initial equation system. Completeness is more difficult since we have to take into account all possible transformations. Therefore we will show that a solvable equation system can be transformed into another very simple equation system which has “an obvious solution”. Such a final equation system is called in “solved form”. According to [11, 21] we call an equation $x \approx t \in E$ solved (in E) if x is a variable which occurs neither in t nor anywhere else in E . In this case variable x is also called solved (in E). An equation system is solved or in solved form if all its equations are solved. A variable or equation is unsolved in E if it occurs in E but is not solved.

The lazy unification calculus in the present form cannot transform each solvable equation system into a solved form since equations between variables are not simplified. For instance, the equation system

$$x \approx f(y), y \approx z_1, y \approx z_2, z_1 \approx z_2$$

is irreducible w.r.t. $\xrightarrow{\text{lu}}$ but not in solved form since the variables y, z_1, z_2 have multiple occurrences. Fortunately, this is not a problem since a solution can be extracted by merging the variables occurring in unsolved equations. Therefore we call this system quasi-solved. An equation system is quasi-solved if each equation $s \approx t$ is solved or has the property $s, t \in \mathcal{X}$. In the following we will show that a quasi-solved equation system has solutions which can be easily computed by applying the rules in Figure 2 to it. The separation between the lazy unification rules in Figure 1 and the variable elimination rules in Figure 2 has technical reasons that will become apparent later (e.g., applying variable elimination to the

equation $y \approx z_1$ may not reduce the complexity measure used in our completeness proofs). However, it is obvious to obtain the solutions of a quasi-solved equation system E . For this purpose we transform E by the rules in Figure 2 into a solved equation system which has a direct solution. This is always possible because \xrightarrow{var} is terminating, preserves solutions, and transforms each quasi-solved system into a solved one (see [17] for details). Moreover, the solutions of an equation system in solved form can be obtained as follows:

Proposition 1. *Let $E = \{x_1 \approx t_1, \dots, x_n \approx t_n\}$ be an equation system in solved form. Then the substitution set*

$$\{\gamma \circ \{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\} \mid \gamma \text{ is a ground substitution}\}$$

is a complete set of solutions for E .

Therefore it is sufficient to transform an equation system into a quasi-solved form. The soundness of the lazy unification calculus is implied by the following theorem which can be proved by a case analysis on the applied transformation rule [17].

Theorem 2. *Let E and E' be equation systems with $E \xrightarrow{lu} E'$. Then each solution σ of E' is also a solution of E .*

For the completeness we show that for each solution of an equation system there is a derivation into a quasi-solved form that has the same solution. Note that the solution of the quasi-solved form cannot be identical to the required solution since new additional variables are generated during the derivation (by lazy narrowing and partial binding transformations). But this is not a problem since we are interested in solutions w.r.t. the variables of the initial equation system.

Theorem 3. *Let E be a solvable equation system with solution σ . Then there exists a derivation $E \xrightarrow{lu}^* E'$ with E' in quasi-solved form such that E' has a solution σ' with $\sigma'(x) =_{\mathcal{R}} \sigma(x)$ for all $x \in \text{Var}(E)$.*

Proof. We show the existence of a derivation from E into a quasi-solved equation system by the following steps:

1. We define a reduction relation \Rightarrow on pairs of the form (σ, E) (where E is an equation system and σ is a solution of E) with the property that $(\sigma, E) \Rightarrow (\sigma', E')$ implies $E \xrightarrow{lu} E'$ and $\sigma'(x) = \sigma(x)$ for all $x \in \text{Var}(E)$.
2. We define a terminating ordering \succ on these pairs.
3. We show: If E has a solution σ but E is not in quasi-solved form, then there exists a pair (σ', E') with $(\sigma, E) \Rightarrow (\sigma', E')$ and $(\sigma, E) \succ (\sigma', E')$.

2 and 3 implies that each solvable equation system can be transformed into a quasi-solved form. By 1, the solution of this quasi-solved form is the required solution of the initial equation system.

In the sequel we will show 1 and 3 in parallel. First we define the terminating ordering \succ . For this purpose we use the *strict subterm ordering* \succ_{sst} on terms defined by $t \succ_{sst} s$ iff there is a position p in t with $t|_p = s \neq t$. Since R is a terminating term rewriting system, the relation \rightarrow_R on terms is also terminating. Let \gg be the transitive closure of the relation $\rightarrow_R \cup \succ_{sst}$. Then \gg is also terminating [20].⁵ Now we define the following ordering on pairs (σ, E) : $(\sigma, E) \succ (\sigma', E')$ iff $\{\sigma(s), \sigma(t) \mid s \approx t \in E \text{ unsolved}\} \gg_{mul} \{\sigma'(s'), \sigma'(t') \mid s' \approx t' \in E' \text{ unsolved}\}$ (*)

⁵ Note that the use of the relation \rightarrow_R instead of \gg (as done in [6]) is not sufficient for the completeness proof since \rightarrow_R has not the subterm property [4] in general.

where \gg_{mul} is the multiset extension⁶ of the ordering \gg (all sets in this definition are multisets). \gg_{mul} is terminating (note that all multisets considered here are finite) since \gg is terminating [4].

Now we will show that we can apply a transformation step to a solvable but unsolved equation system such that its complexity is reduced. Let E be an equation system not in quasi-solved form and σ be a solution of E . Since E is not quasi-solved, there must be an equation which has one of the following forms:

1. There is an equation $E = s \approx t, E_0$ with $s, t \notin \mathcal{X}$: Let $s = f(s_1, \dots, s_n)$ with $n \geq 0$ (the other case is symmetric). Consider an innermost derivation of the normal forms of $\sigma(s)$ and $\sigma(t)$:

- (a) No rewrite step is performed at the root of $\sigma(s)$ and $\sigma(t)$: Then t has the form $t = f(t_1, \dots, t_n)$ and $\sigma(s) \downarrow_{\mathcal{R}} = \sigma(t) \downarrow_{\mathcal{R}} = f(u_1, \dots, u_n)$. Since $\sigma(s)$ and $\sigma(t)$ are not reducible at the root, $\sigma(s_i) \downarrow_{\mathcal{R}} = u_i = \sigma(t_i) \downarrow_{\mathcal{R}}$ for $i = 1, \dots, n$. Now we apply the decomposition transformation and obtain the equation system

$$E' = s_1 \approx t_1, \dots, s_n \approx t_n, E_0$$

Obviously, σ is a solution of E' . Moreover, the complexity of the new equation system is reduced because the equation $s \approx t$ is unsolved in E and each $\sigma(s_i)$ and $\sigma(t_i)$ is smaller than $\sigma(s)$ and $\sigma(t)$, respectively, since \gg contains the strict subterm ordering \succ_{sst} . Hence $(\sigma, E) \succ (\sigma, E')$.

- (b) A rewrite step is performed at the root of $\sigma(s)$, i.e., the innermost rewriting sequence of $\sigma(s)$ has the form

$$\sigma(s) \rightarrow_{\mathcal{R}}^* f(s'_1, \dots, s'_n) \rightarrow_{\mathcal{R}} \theta(r) \rightarrow_{\mathcal{R}}^* \sigma(s) \downarrow_{\mathcal{R}}$$

where $f(l_1, \dots, l_n) \rightarrow r$ is a new variant of a rewrite rule, $\theta(l_i) = s'_i$ and $\sigma(s_i) \rightarrow_{\mathcal{R}}^* s'_i$ for $i = 1, \dots, n$. An application of the lazy narrowing transformation yields the equation system

$$E' = s_1 \approx l_1, \dots, s_n \approx l_n, r \approx t, E_0$$

We extend σ to a new substitution σ' with $\sigma'(x) = \theta(x)$ for all $x \in \text{Dom}(\theta)$ (this is always possible since θ does only work on the variables of the new variant of the rewrite rule). σ' is a solution of E' since

$$\sigma'(s_i) = \sigma(s_i) \rightarrow_{\mathcal{R}}^* s'_i = \theta(l_i) = \sigma'(l_i)$$

and

$$\sigma'(r) = \theta(r) \rightarrow_{\mathcal{R}}^* \sigma(s) \downarrow_{\mathcal{R}} \leftrightarrow_{\mathcal{R}}^* \sigma(t) = \sigma'(t)$$

Since the transitive closure of $\rightarrow_{\mathcal{R}}$ is contained in \gg , $\sigma(s_i) \gg \sigma'(l_i)$ (if $\sigma(s_i) \neq \sigma'(l_i)$) and $\sigma(s) \gg \sigma'(r)$. Since $s \approx t$ is unsolved in E , the term $\sigma(s)$ is contained in the left multiset of the ordering definition (*), and it is replaced by the smaller terms $\sigma(s_1), \dots, \sigma(s_n), \sigma'(l_1), \dots, \sigma'(l_n), \sigma'(r)$ ($\sigma(s) \gg \sigma(s_i)$ since \gg contains the strict subterm ordering). Therefore the new equation system is smaller w.r.t. \succ , i.e., $(\sigma, E) \succ (\sigma', E')$.

2. There is an equation $E = x \approx t, E_0$ with $t = f(t_1, \dots, t_n)$ and x unsolved in E : Hence $x \in \text{Var}(t) \cup \text{Var}(E_0)$. Again, we consider an innermost derivation of the normal form of $\sigma(t)$:

- (a) A rewrite step is performed at the root of $\sigma(t)$. Then we apply a lazy narrowing step and proceed as in the previous case.

⁶ The multiset ordering \gg_{mul} is the transitive closure of the replacement of an element by a finite number of elements that are smaller w.r.t. \gg [4].

- (b) No rewrite step is performed at the root of $\sigma(t)$, i.e., $\sigma(t)\downarrow_{\mathcal{R}} = f(t'_1, \dots, t'_n)$ and $\sigma(t_i)\downarrow_{\mathcal{R}} = t'_i$ for $i = 1, \dots, n$. We apply the partial binding transformation and obtain the equation system

$$E' = x \approx f(x_1, \dots, x_n), x_1 \approx \phi(t_1), \dots, x_n \approx \phi(t_n), \phi(E_0)$$

where $\phi = \{x \mapsto f(x_1, \dots, x_n)\}$ and x_i are new variables. We extend σ to a substitution σ' by adding the bindings $\sigma'(x_i) = t'_i$ for $i = 1, \dots, n$. Then

$$\sigma'(f(x_1, \dots, x_n)) = f(t'_1, \dots, t'_n) = \sigma(t)\downarrow_{\mathcal{R}} \leftrightarrow_{\mathcal{R}}^* \sigma(t) \leftrightarrow_{\mathcal{R}}^* \sigma(x) = \sigma'(x)$$

Moreover, $\sigma'(\phi(x)) = \sigma'(x)\downarrow_{\mathcal{R}}$ which implies $\sigma'(s) \leftrightarrow_{\mathcal{R}}^* \sigma'(\phi(s))$ for all terms s . Hence $\sigma'(\phi(t_i)) \leftrightarrow_{\mathcal{R}}^* \sigma'(t_i) \leftrightarrow_{\mathcal{R}}^* t'_i = \sigma'(x_i)$. Altogether, σ' is a solution of E' .

It remains to show that this transformation reduces the complexity of the equation system. Since $\sigma'(\phi(x)) = \sigma(x)\downarrow_{\mathcal{R}}$, we have $\sigma(x) \rightarrow_{\mathcal{R}}^* \sigma'(\phi(x))$. Hence $\sigma(E_0)$ is equal to $\sigma'(\phi(E_0))$ (if $\sigma(x) = \sigma'(\phi(x))$) or $\sigma'(\phi(E_0))$ is smaller w.r.t. \succ_{mul} . Therefore it remains to check that $\sigma(t)$ is greater than each $\sigma'(x_1), \dots, \sigma'(x_n), \sigma'(\phi(t_1)), \dots, \sigma'(\phi(t_n))$ w.r.t. \succ (note that the equation $x \approx t$ is unsolved in E , but the equation $x \approx f(x_1, \dots, x_n)$ is solved in E'). First of all, $\sigma(t) \succ \sigma(t_i)$ since \succ includes the strict subterm ordering. Moreover, $\sigma(t_i) \rightarrow_{\mathcal{R}}^* \sigma'(x_i)$, i.e., $\sigma'(x_i)$ is equal or smaller than $\sigma(t_i)$ w.r.t. \succ for $i = 1, \dots, n$. This implies $\sigma(t) \succ \sigma'(x_i)$. Similarly, $\sigma'(\phi(t_i))$ is equal or smaller than $\sigma(t_i)$ w.r.t. \succ since $\sigma'(\phi(x)) = \sigma(x)\downarrow_{\mathcal{R}}$. Thus $\sigma(t) \succ \sigma'(\phi(t_i))$. Altogether, $(\sigma, E) \succ (\sigma', E')$. \square

We want to point out that there exist also other orderings on substitution/equation system pairs to prove the completeness of our calculus. However, the ordering chosen in the above proof is tailored to a simple proof for the completeness of lazy unification with simplification as we will see in the next section.

The results of this section imply that a complete set of solutions for a given equation system E can be computed by enumerating all derivations in the lazy unification calculus from E into a quasi-solved equation system. Due to the nondeterminism in the lazy unification calculus, there are many unsuccessful and often infinite derivations. Therefore we will show in the next section how to reduce this nondeterminism by integrating a deterministic simplification process into the lazy unification calculus. More determinism can be achieved by dividing the set of function symbols into constructors and defined functions. This will be the subject of Section 5.

4 Integrating simplification into lazy unification

The lazy unification calculus admits a high degree of nondeterminism even if there is only one reasonable derivation. This is due to the fact that functional expressions are processed “too lazy”.

Example 3. Consider the rewrite rules

$$\begin{array}{ll} f(a) \rightarrow c & g(a) \rightarrow a \\ f(b) \rightarrow d & g(b) \rightarrow b \end{array}$$

and the equation $f(g(b)) \approx d$. Then there are four different derivations in our lazy unification calculus, but only one derivation is successful. If we would first compute the normal form of $f(g(b))$, which is d , then there is only one possible derivation: $d \approx d \xrightarrow{lu} \emptyset$. Hence we will show that the lazy unification calculus remains to be sound and complete if the (deterministic!) normalization of terms is included. \square

It is well-known [9, 16] that the inclusion of inductive consequences for normalization may have an essential effect on the search space reduction in normalizing narrowing strategies. Therefore we will also allow the use of additional inductive consequences for normalization. A rewrite rule $l \rightarrow r$ is called *inductive consequence* (of \mathcal{R}) if $\sigma(l) =_{\mathcal{R}} \sigma(r)$ for all ground substitutions σ . For instance, the rule $x + 0 \rightarrow x$ is an inductive consequence of the term rewriting system

$$0 + y \rightarrow y \quad s(x) + y \rightarrow s(x + y)$$

If we want to solve the equation $s(x) + 0 \approx s(x)$, our basic lazy unification calculus would enumerate the solutions $x \mapsto 0$, $x \mapsto s(0)$, $x \mapsto s(s(0))$ and so on, i.e., this equation has an infinite search space. Using the inductive consequence $x + 0 \rightarrow x$ for normalization, the equation $s(x) + 0 \approx s(x)$ is reduced to $s(x) \approx s(x)$ and then transformed into the quasi-solved form $x \approx x$ representing the solution set where x is replaced by any ground term.⁷

In the following we assume that \mathcal{S} is a set of inductive consequences of \mathcal{R} (the set of *simplification rules*) so that the rewrite relation $\rightarrow_{\mathcal{S}}$ is terminating. We will use rules from \mathcal{R} for lazy narrowing and rules from \mathcal{S} for simplification. Note that each rule from \mathcal{R} is also an inductive consequence and can be included in \mathcal{S} . But we do not require that all rules from \mathcal{R} must be used for normalization. This is reasonable if there are duplicating rules where one variable of the left-hand side occurs several times on the right-hand side, like $f(x) \rightarrow g(x, x)$. If we normalize the equation $f(s) \approx t$ with this rule, then the term s is duplicated which may increase the computational costs if the evaluation of s is necessary and costly. In such a case it would be better to use this rule only in lazy narrowing steps.

In order to include simplification into the lazy unification calculus, we define a relation $\Rightarrow_{\mathcal{S}}$ on systems of equations. $s \approx t \Rightarrow_{\mathcal{S}} s' \approx t'$ iff s' and t' are normal forms of s and t w.r.t. $\rightarrow_{\mathcal{S}}$, respectively. $E \Rightarrow_{\mathcal{S}} E'$ iff $E = e_1, \dots, e_n$ and $E' = e'_1, \dots, e'_n$ where $e_i \Rightarrow_{\mathcal{S}} e'_i$ for $i = 1, \dots, n$. Note that $\Rightarrow_{\mathcal{S}}$ describes a deterministic computation process.⁸ $E \xrightarrow{\text{luz}} E'$ is a derivation step in the *lazy unification calculus with simplification* if $E \Rightarrow_{\mathcal{S}} \overline{E} \xrightarrow{\text{luz}} E'$ for some \overline{E} .

The soundness of the calculus $\xrightarrow{\text{luz}}$ can be shown by a simple induction on the computation steps using Theorem 2 and the following lemma which shows the soundness of one rewrite step with a simplification rule:

Lemma 4. *Let $s \approx t$ be an equation and $s \rightarrow_{\mathcal{S}} s'$ be a rewrite step. Then each solution of $s' \approx t$ is also a solution of $s \approx t$.*

For the completeness proof we have to show that solutions are not lost by the application of inductive consequences:

Lemma 5. *Let E be an equation system and σ be a solution of E . If $E \Rightarrow_{\mathcal{S}} E'$, then σ is a solution of E' .*

⁷ In larger single-sorted term rewriting systems it would be difficult to find inductive consequences. E.g., $x + 0 \rightarrow x$ is not an inductive consequence if there is a constant a since $a + 0 =_{\mathcal{R}} a$ is not valid. However, in practice specifications are many-sorted and then inductive consequences must be valid only for all well-sorted ground substitutions. Therefore we want to point out that all results in this paper can also be extended to many-sorted term rewriting systems in a straightforward way.

⁸ If there exist more than one normal form w.r.t. $\rightarrow_{\mathcal{S}}$, it is sufficient to select *don't care* one of these normal forms.

This lemma would imply the completeness of the calculus \xrightarrow{lus} if a derivation step with \Rightarrow_S does not increase the ordering used in the proof of Theorem 3. Unfortunately, this is not the case in general since the termination of \rightarrow_R and \rightarrow_S may be based on different orderings (e.g., $R = \{a \rightarrow b\}$ and $S = \{b \rightarrow a\}$). In order to avoid such problems, we require that the relation $\rightarrow_{R \cup S}$ is terminating which is not a real restriction in practice.

Theorem 6. *Let S be a set of inductive consequences of the ground confluent and terminating term rewriting system R such that $\rightarrow_{R \cup S}$ is terminating. Let E be a solvable equation system with solution σ . Then there exists a derivation $E \xrightarrow{lus}^* E'$ such that E' is in quasi-solved form and has a solution σ' with $\sigma'(x) =_R \sigma(x)$ for all $x \in \text{Var}(E)$.*

Proof. In the proof of Theorem 3 we have shown how to apply a transformation step to an equation system not in quasi-solved form such that the solution is preserved. We can use the same proof for the transformation \xrightarrow{lus} since Lemma 5 shows that normalization steps preserve solutions. The only difference concerns the ordering where we use $\rightarrow_{R \cup S}$ instead of \rightarrow_R , i.e., \gg is now defined to be the transitive closure of the relation $\rightarrow_{R \cup S} \cup \succ_{sst}$. Clearly, this does not change anything in the proof of Theorem 3. Moreover, the relation \Rightarrow_S does not increase the complexity w.r.t. this ordering but reduces it if inductive consequences are applied since \rightarrow_S is contained in \gg . \square

These results show that we can integrate the deterministic simplification process into the lazy unification calculus without loosing soundness and completeness. Note that the rules from S can only be applied if their left-hand sides can be matched with a subterm of the current equation system. If these subterms are not sufficiently instantiated, the rewrite rules are not applicable and hence we loose potential determinism in the unification process.

Example 4. Consider the rules

$$\text{zero}(s(x)) \rightarrow \text{zero}(x) \quad \text{zero}(0) \rightarrow 0$$

(assume that these rules are contained in R as well as in S) and the equation system $\text{zero}(x) \approx 0, x \approx 0$. Then there exists the following derivation in our calculus (this derivation is also possible in the unification calculi in [11, 22]):

$$\begin{aligned} & \text{zero}(x) \approx 0, x \approx 0 \\ \xrightarrow{lus} & x \approx s(x_1), \text{zero}(x_1) \approx 0, x \approx 0 && (\text{lazy narrowing}) \\ \xrightarrow{lus} & x \approx s(x_1), x_1 \approx s(x_2), \text{zero}(x_2) \approx 0, x \approx 0 && (\text{lazy narrowing}) \\ \xrightarrow{lus} & \dots \end{aligned}$$

This infinite derivation could be avoided if we apply the partial binding rule in the first step:

$$\begin{aligned} \text{zero}(x) \approx 0, x \approx 0 & \xrightarrow{lus} \text{zero}(0) \approx 0, x \approx 0 && (\text{partial binding}) \\ & \Rightarrow_S 0 \approx 0, x \approx 0 && (\text{rewriting with second rule}) \\ & \xrightarrow{lus} x \approx 0 && (\text{decomposition}) \end{aligned}$$

In the next section we will present an optimization which prefers the latter derivation and avoids the first infinite derivation. \square

Decomposition of constructor equations

$$c(t_1, \dots, t_n) \approx c(t'_1, \dots, t'_n), E \xrightarrow{luc} t_1 \approx t'_1, \dots, t_n \approx t'_n, E \quad \text{if } c \in \mathcal{C}$$

Full binding of variables to ground constructor terms

$$x \approx t, E \xrightarrow{luc} x \approx t, \phi(E) \quad \text{if } x \in \text{Var}(E), t \in T(\mathcal{C}, \emptyset) \text{ and } \phi = \{x \mapsto t\}$$

Partial binding of variables to constructor terms

$$x \approx c(t_1, \dots, t_n), E \xrightarrow{luc} x \approx c(x_1, \dots, x_n), x_1 \approx \phi(t_1), \dots, x_n \approx \phi(t_n), \phi(E)$$

if $x \in \text{Var}(c(t_1, \dots, t_n)) \cup \text{Var}(E)$, $x \notin \text{cvar}(c(t_1, \dots, t_n))$ and $\phi = \{x \mapsto c(x_1, \dots, x_n)\}$
(x_i new variable)

Figure 3. Deterministic transformations for constructor-based rewrite systems

5 Constructor-based systems

In practical applications of equational logic programming a distinction is made between operation symbols to construct data terms, called *constructors*, and operation symbols to operate on data terms, called *defined functions* (see, for instance, the functional logic languages ALF [15], BABEL [23], K-LEAF [13], SLOG [9], or the RAP system [12]). Such a distinction allows to optimize our unification calculus. Therefore we assume in this section that the signature \mathcal{F} is divided into two sets $\mathcal{F} = \mathcal{C} \cup \mathcal{D}$, called constructors and defined functions, with $\mathcal{C} \cap \mathcal{D} = \emptyset$. A *constructor term* t is built from constructors and variables, i.e., $t \in T(\mathcal{C}, \mathcal{X})$. The distinction between constructors and defined functions comes with the restriction that for all rewrite rules $l \rightarrow r$ the outermost symbol of l is always a defined function.

The important property of such constructor-based term rewriting systems is the irreducibility of constructor terms. Due to this fact we can specialize the rules of our basic lazy unification calculus. Therefore we define the deterministic transformations in Figure 3. *Deterministic transformations* are intended to be applied as long as possible before any transformation \xrightarrow{luc} is used. Hence they can be integrated into the deterministic normalization process $\Rightarrow_{\mathcal{S}}$. It is obvious that this modification preserves soundness and completeness. The decomposition transformation for constructor equations must be applied in any case in order to obtain a quasi-solved equation system since a lazy narrowing step \mathcal{R} cannot be applied to constructor equations. The full binding of variables to ground constructor terms is an optimization which combines subsequent applications of partial binding transformations. This transformation decreases the complexity used in the proof of Theorem 6 since a constructor term is always in normal form. The partial binding transformation for constructor terms performs an eager (partial) binding of variables to constructor terms since a lazy narrowing step cannot be applied to the constructor term. Moreover, this binding transformation is combined with an *occur check* since it cannot be applied if $x \in \text{cvar}(c(t_1, \dots, t_n))$ where cvar denotes the set of all variables occurring outside terms headed by defined function symbols. This restriction avoids infinite derivations of the following kind:

$$\begin{aligned} x \approx c(x) &\xrightarrow{luc} x \approx c(x_1), x_1 \approx c(x_1) && (\text{partial binding}) \\ &\xrightarrow{luc} x \approx c(x_1), x_1 \approx c(x_2), x_2 \approx c(x_2) && (\text{partial binding}) \\ &\xrightarrow{luc} \dots \end{aligned}$$

A further optimization can be added if all functions are reducible on ground constructor terms, i.e., for all $f \in \mathcal{D}$ and $t_1, \dots, t_n \in T(\mathcal{C}, \emptyset)$ there exists a term t

Clash	$c(t_1, \dots, t_n) \approx d(t'_1, \dots, t'_m), E \xrightarrow{luc} \text{FAIL}$	if $c, d \in \mathcal{C}$ and $c \neq d$
Occur check	$x \approx c(t_1, \dots, t_n), E \xrightarrow{luc} \text{FAIL}$	if $x \in cvar(c(t_1, \dots, t_n))$

Figure 4. Failure rules for constructor-based rewrite systems

with $f(t_1, \dots, t_n) \rightarrow_{\mathcal{R}} t$. In this case all ground terms have a ground constructor normal form and therefore the partial binding transformation of \xrightarrow{luc} can be completely omitted which increases the determinism in the lazy unification calculus.

If we invert the deterministic transformation rules, we obtain a set of failure rules shown in Figure 4. *Failure rules* are intended to be tried during the deterministic transformations. If a failure rule is applicable, the derivation can be safely terminated since the equation system cannot be transformed into a quasi-solved system.

6 Examples

In this section we demonstrate the improved computational power of our lazy unification calculus with simplification by means of two examples. The first example shows that simplification reduces the search space in the presence of rewrite rules with overlapping left-hand sides.

Example 5. Consider the following ground confluent and terminating rewrite system defining the Boolean operator \vee and the predicate *even* on natural numbers:

$$\begin{array}{ll} \text{true} \vee b \rightarrow \text{true} & \text{even}(0) \rightarrow \text{true} \\ b \vee \text{true} \rightarrow \text{true} & \text{even}(s(0)) \rightarrow \text{false} \\ \text{false} \vee \text{false} \rightarrow \text{false} & \text{even}(s(s(x))) \rightarrow \text{even}(x) \end{array}$$

If we want to solve the equation $\text{even}(z) \vee \text{true} \approx \text{true}$, the lazy unification calculus without simplification could apply a lazy narrowing step with the first \vee -rule. This yields the equation system

$$\text{even}(z) \approx \text{true}, \text{true} \approx b, \text{true} \approx \text{true}$$

Now there are infinitely many solutions to the new equation $\text{even}(z) \approx \text{true}$ by instantiating the variable z with the values $s^{2*i}(0)$, $i \geq 0$, i.e., the lazy unification calculus without simplification (cf. Section 3) has an infinite search space. The same is true for other lazy unification calculi [11, 22] or lazy narrowing calculi [23, 27]. Moreover, in a sequential implementation of lazy narrowing by backtracking [14] only an infinite set of specialized solutions would be computed without ever trying the second \vee -rule. But if we use our lazy unification calculus with simplification where all rewrite rules are used for simplification (i.e., $\mathcal{R} = \mathcal{S}$), then the initial equation $\text{even}(z) \vee \text{true} \approx \text{true}$ is first simplified to $\text{true} \approx \text{true}$ by rewriting with the second \vee -rule. Hence our calculus has a finite search space. \square

If the left-hand sides of the rewrite rules do not overlap, i.e., if the functions are defined by a case distinction on one argument, then there exists a lazy narrowing strategy (*needed narrowing* [1]) which is optimal w.r.t. the length of derivations. However, unsuccessful infinite derivations can be avoided also in this case by our lazy unification calculus with simplification if inductive consequences are added to the set of simplification rules.

Example 6. Consider the following rewrite rules for the addition and multiplication on natural numbers where $\mathcal{C} = \{0, s\}$ are constructors and $\mathcal{D} = \{+, *\}$ are defined functions:

$$\begin{array}{ll} 0 + y \rightarrow y & (1) \\ s(x) + y \rightarrow s(x + y) & (2) \end{array} \quad \begin{array}{ll} 0 * y \rightarrow 0 & (3) \\ s(x) * y \rightarrow y + x * y & (4) \end{array}$$

If we use this confluent and terminating set of rewrite rules for lazy narrowing (\mathcal{R}) as well as for normalization (\mathcal{S}) and add the inductive consequence $x * 0 \rightarrow 0$ to \mathcal{S} , then our lazy unification calculus with simplification has a finite search space for the equation $x * y \approx s(0)$. This is due to the fact that the following derivation can be terminated using the inductive consequence and the clash rule:

$$\begin{aligned} x * y &\approx s(0) \\ \xrightarrow{\text{lu}} x &\approx s(x_1), y \approx y_1, y_1 + x_1 * y_1 \approx s(0) && (\text{lazy narrowing, rule 4}) \\ \xrightarrow{\text{lu}} x &\approx s(x_1), y \approx y_1, y_1 \approx 0, x_1 * y_1 \approx y_2, y_2 \approx s(0) && (\text{lazy narrowing, rule 1}) \\ \xrightarrow{\text{lu}\zeta} x &\approx s(x_1), y \approx 0, y_1 \approx 0, x_1 * 0 \approx y_2, y_2 \approx s(0) && (\text{bind variable } y_1) \\ \xrightarrow{\text{lu}\zeta} x &\approx s(x_1), y \approx 0, y_1 \approx 0, x_1 * 0 \approx s(0), y_2 \approx s(0) && (\text{bind variable } y_2) \\ \Rightarrow_{\mathcal{S}} x &\approx s(x_1), y \approx 0, y_1 \approx 0, 0 \approx s(0), y_2 \approx s(0) && (\text{reduce } x_1 * 0) \\ \xrightarrow{\text{lu}\zeta} \text{FAIL} & && (\text{clash between 0 and } s) \end{aligned}$$

The equation $x_1 * 0 \approx s(0)$ could not be transformed into the equation $0 \approx s(0)$ without the inductive consequence. Consequently, an infinite derivation would occur in our basic unification calculus of Section 3.

Note that other lazy unification calculi [11, 22] or lazy narrowing calculi [23, 27] have an infinite search space for this equation. It is also interesting to note that a normalizing innermost narrowing strategy as in [9] has also an infinite search space even if the same inductive consequences are available. This shows the advantage of combining a lazy strategy with a simplification process. \square

7 Conclusions

In this paper we have presented a calculus for unification in the presence of an equational theory. In order to obtain a small search space, the calculus is designed in the spirit of lazy evaluation, i.e., functions are not evaluated if their result is not required to solve the unification problem. The most important property of our calculus is the inclusion of a deterministic simplification process. This has the positive effect that our calculus is more efficient (in terms of the search space size) than other lazy unification calculi or eager narrowing calculi (like basic narrowing, innermost narrowing) with simplification. We think that our calculus is the basis of efficient implementations of future functional logic languages.

Acknowledgements. The author is grateful to Harald Ganzinger for his pointer to a suitable termination ordering and to two anonymous referees for their helpful remarks. The research described in this paper was supported in part by the German Ministry for Research and Technology (BMFT) under grant ITS 9103 and by the ESPRIT Basic Research Working Group 6028 (Construction of Computational Logics).

References

1. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. In *Proc. 21st ACM Symp. on Principles of Programming Languages*, pp. 268–279, Portland, 1994.
2. D. Bert and R. Echahed. Design and Implementation of a Generic, Logic and Functional Programming Language. In *Proc. ESOP'86*, pp. 119–132. Springer LNCS 213, 1986.
3. J. Darlington and Y. Guo. Narrowing and unification in functional programming – an evaluation mechanism for absolute set abstraction. In *Proc. of the Conference on Rewriting Techniques and Applications*, pp. 92–108. Springer LNCS 355, 1989.

4. N. Dershowitz. Termination of Rewriting. *J. Symbolic Computation*, Vol. 3, pp. 69–116, 1987.
5. N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, Vol. B, pp. 243–320. Elsevier, 1990.
6. N. Dershowitz, S. Mitra, and G. Sivakumar. Equation Solving in Conditional AC-Theories. In *Proc. ALP'90*, pp. 283–297. Springer LNCS 463, 1990.
7. R. Echahed. Uniform Narrowing Strategies. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pp. 259–275. Springer LNCS 632, 1992.
8. M.J. Fay. First-Order Unification in an Equational Theory. In *Proc. 4th Workshop on Automated Deduction*, pp. 161–167, Austin (Texas), 1979. Academic Press.
9. L. Fribourg. SLOG: A Logic Programming Language Interpreter Based on Clausal Superposition and Rewriting. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 172–184, Boston, 1985.
10. J.H. Gallier and S. Raatz. Extending SLD-Resolution to Equational Horn Clauses Using E-Unification. *Journal of Logic Programming* (6), pp. 3–43, 1989.
11. J.H. Gallier and W. Snyder. Complete Sets of Transformations for General E-Unification. *Theoretical Computer Science*, Vol. 67, pp. 203–260, 1989.
12. A. Geser and H. Hussmann. Experiences with the RAP system – a specification interpreter combining term rewriting and resolution. In *Proc. ESOP 86*, pp. 339–350. Springer LNCS 213, 1986.
13. E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel LEAF: A Logic plus Functional Language. *Journal of Computer and System Sciences*, Vol. 42, No. 2, pp. 139–185, 1991.
14. W. Hans, R. Loogen, and S. Winkler. On the Interaction of Lazy Evaluation and Backtracking. In *Proc. PLILP'92*, pp. 355–369. Springer LNCS 631, 1992.
15. M. Hanus. Compiling Logic Programs with Equality. In *Proc. PLILP'90*, pp. 387–401. Springer LNCS 456, 1990.
16. M. Hanus. Improving Control of Logic Programs by Using Functional Logic Languages. In *Proc. PLILP'92*, pp. 1–23. Springer LNCS 631, 1992.
17. M. Hanus. Lazy Unification with Inductive Simplification. Technical Report MPI-I-93-215, Max-Planck-Institut für Informatik, Saarbrücken, 1993.
18. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *To appear in Journal of Logic Programming*, 1994.
19. S. Hölldobler. *Foundations of Equational Logic Programming*. Springer LNCS 353, 1989.
20. J.-P. Jouannaud and H. Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal on Computing*, Vol. 15, No. 4, pp. 1155–1194, 1986.
21. A. Martelli and U. Montanari. An Efficient Unification Algorithm. *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 2, pp. 258–282, 1982.
22. A. Martelli, G.F. Rossi, and C. Moiso. Lazy Unification Algorithms for Canonical Rewrite Systems. In Hassan Aït-Kaci and Maurice Nivat, editors, *Resolution of Equations in Algebraic Structures, Volume 2, Rewriting Techniques*, chapter 8, pp. 245–274. Academic Press, New York, 1989.
23. J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: The Language BABEL. *Journal of Logic Programming*, Vol. 12, pp. 191–223, 1992.
24. W. Nutt, P. Réty, and G. Smolka. Basic Narrowing Revisited. *Journal of Symbolic Computation*, Vol. 7, pp. 295–317, 1989.
25. P. Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1988.
26. G.D. Plotkin. Building-in Equational Theories. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, pp. 73–90, 1972.
27. U.S. Reddy. Narrowing as the Operational Semantics of Functional Languages. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 138–151, Boston, 1985.
28. P. Réty. Improving basic narrowing techniques. In *Proc. of the Conference on Rewriting Techniques and Applications*, pp. 228–241. Springer LNCS 256, 1987.
29. J.H. Siekmann. An Introduction to Unification Theory. In *Formal Techniques in Artificial Intelligence*, pp. 369–425. Elsevier Science Publishers, 1990.

Polymorphic Binding-Time Analysis

Fritz Henglein & Christian Mossin*

DIKU, Department of Computer Science
University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen Ø
Denmark
e-mail: henglein@diku.dk & mossin@diku.dk

Abstract. Binding time analysis is an important part of off-line partial evaluation, annotating expressions as being safely evaluable from known data or possibly depending on unknown data. Most binding-time analyses have been *monovariant*, allowing only one binding-time description for each function. The idea of *polyvariance* is to allow multiple binding time descriptions of each function, by duplicating the function [6, 2] or by associating a set of binding time descriptions to each function [3]. Instead we present an inference based binding time analysis polymorphic in binding time values. This polymorphism captures a very powerful notion of polyvariance limited only by the (standard) types of the language. Polymorphism gives a much simpler definition than the known polyvariant schemes allowing us to reason formally about the system and prove it correct.

This paper is based on work in [14].

1 Introduction

In *binding-time analysis* input is divided into *static* (denoted S) and *dynamic* (denoted D). The job of binding-time analysis is to find out which operations and program phrases can be executed with only static inputs available. Note that the analysis has no knowledge of the actual *values* of the static inputs, only *that* they will be available before the dynamic inputs.

The result of a binding-time analysis can be used to guide the actions of both on-line and off-line partial evaluators. In either case operations classified as static are guaranteed to be executable by the partial evaluator once the static inputs are available. In an on-line partial evaluator the remaining operations, classified as dynamic, require (specialization time) checking to determine whether an operation can be executed or must be deferred to run-time. In an off-line partial evaluator the dynamic operations are automatically deferred to run-time. Thus an off-line partial evaluator is generally more efficient than its on-line counterpart, at the expense of deferring more operations to run-time.²

* Supported by the Danish Technical Research Council

² While binding-time analysis seems to have been applied exclusively in off-line partial evaluators this shows that binding-time analysis can be employed with advantage by on-line partial evaluators since it eliminates some checking actions.

In many existing partial evaluators (*e.g.* Similix [1]) every user-defined function is assigned exactly one binding time description. This description has to be a safe approximation to all calls to the function, so if the function is called with actual parameters of different binding times (*e.g.* with (S,D) and (D,S)), the description of the function can only be a “widened” binding time, in this example (D,D). A *polyvariant* binding-time analysis seeks to remedy this problem by keeping function calls made in different binding-time contexts separate.

We avoid widening by generalizing type-based monovariant binding-time analysis to a form of *polymorphism* in binding times by adding explicit *abstraction* over binding-time parameters to our binding-time descriptions. Function definitions are *parameterized* over binding times instead of committing these to be S or D. This allows keeping the binding times of the calls to a function separate by instantiating its binding-time parameters to different binding times at the call sites.

2 New Results

We present a polyvariant binding-time analysis for an ML-like language that extends a type-based monovariant binding-time analysis by explicit polymorphism over binding-time parameters. We show the following:

- The type system has the *principal typing* property. This guarantees that every program has a (parameterized) binding-time annotation that subsumes all others for the same program; *i.e.*, it can be used in any context where the program could occur. This admits modular (“local”) binding-time analysis of a (function) definition, independent of any of its applications.
- Explicit binding-time application can be interpreted by a specializer as ordinary parameter passing.
- The result of binding-time analysis is an annotation of the *original source* program, not a transformed program. This allows integration into a programming environment and, specifically, support for *static binding-time debugging* [12]. Binding-time debugging is helpful in improving the *binding-time separation* in a program (by rewriting it) [9, 13].
- The binding-time analysis is proved to be *correct* with respect to a canonical semantics (corresponding to a canonical specializer) for binding-time annotated programs.
- The binding-time analysis supports polymorphism in the binding-times independent of the polymorphism of the type discipline of the language. Specifically, it allows binding-time parameterized types both in let-expressions (nonrecursive definitions) and in fix-expressions (recursive definitions). Since the polymorphism in binding times is restricted by the structure of the underlying types this still gives a decidable type inference system.
- The binding-time analysis is *partially complete*. We show that unfolding a definition — whether recursive or not — cannot improve the results of the analysis.

- The analysis is straightforwardly extended to handle structured data. In particular, there is no interference of polyvariance and partially static structures.

We have not systematically analyzed the size of explicitly annotated programs in relation to the original (explicitly typed) source programs. Experimental results indicate, however, that the principal annotation of an individual function definition rarely has more than 10 binding-time parameters after reduction of binding-time constraint sets [14].

3 Related Work

Most of the previous work on polyvariant binding-time analysis is based on abstract interpretation [10, 11, 8]. These binding-time analyses are polyvariant as they provide detailed and almost exact binding-time information for functions and even higher-order functions. Since they are too detailed and “extensional” in nature — i.e., they carry out the analysis without a record of *how* a binding-time value arises — there is no obvious way of using their results in a partial evaluator.

Rytz and Gengler take a pragmatic approach to polyvariant binding-time analysis: monovariant binding-time analysis is trapped when a function’s binding-time value would need to be “widened”. Instead they duplicate the function’s definition and reiterate both control flow and binding time analysis [6] until no more widening steps occur. Being an extension of Similix’ binding time analysis [1] the analysis handles a higher order functional language, but reiterating the analysis is very expensive and no proofs of correctness are given.

Bulyonkov shows how the copying can be accomplished systematically for a first-order language by rewriting the program to contain variables corresponding to binding time values and using a polyvariant (memoizing) *specializer* [2]. This approach makes the technique independent of a particular partial evaluator, and avoids reiterating the analyses.

Consel combines closure and binding time analysis to avoid reiteration [3]. Also, actual copying is avoided by keeping a *set* of different binding-time descriptions with function definitions. The binding-time analysis is extended to such sets of descriptions. The degree of polyvariance is bounded by a function describing how the set of different binding time descriptions is indexed. It appears that in a typed language, this function can be made precise enough to capture the same degree of polyvariance as we obtain. In the partial evaluator Schism [4] a much simpler version of the analysis is implemented. No estimate of the cost of achieving the high degree of polyvariance is given. In its full generality the analysis is still likely to be very expensive, though.

Type-based binding-time analysis for typed languages originated with the Nielsons’ development of the two-level λ -calculus for simply-typed λ -expressions [16] and Gomard’s inference system for untyped λ -expressions [7]. Their analyses are monovariant, and until recently there was no type-based polyvariant analysis.

Contemporary with our work Consel and Jouvelot have developed a type- and effect-based binding-time analysis similar in spirit to ours [5]. They annotate standard types with binding-time *effects*. These effect-annotated types correspond more or less directly to our binding-time annotated types, though with nested binding-time polymorphism. The resulting binding-time analysis can be seen to be analogous to a “sticky” strictness analysis restricted to the simply typed λ -calculus. The programmer must, however, specify explicitly standard types as well as binding times for all formal parameters. With explicit assumptions for all variables the binding times of expressions are computed automatically. Correctness is formulated relative to an operational semantics geared towards reduction to head-normal form that doubles as a standard and specialization semantics.

4 Language

We will use a monomorphic call-by-name language *Exp* as our source language, which is simple enough to avoid important aspects being blurred by syntactical details, but powerful enough to show the generality of our method. The language syntax (including static semantics) is defined by the type rules of Fig. 1.

	(const)	$A \vdash c : CT(c)$
	(var)	$A \cup \{x:t\} \vdash x:t$
(if)		$\frac{A \vdash e_1 : \text{Bool} \quad A \vdash e_2 : t \quad A \vdash e_3 : t}{A \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t}$
(abstr)		$\frac{A \cup \{x:t\} \vdash e : t_1}{A \vdash \lambda x : t. e : t \rightarrow t_1}$
(appl)		$\frac{A \vdash e_1 : t_2 \rightarrow t_1 \quad A \vdash e_2 : t_2}{A \vdash e_1 @ e_2 : t_1}$
(let)		$\frac{A \vdash e_2 : t_2 \quad A \cup \{x:t_2\} \vdash e_1 : t_1}{A \vdash \text{let } x : t_2 = e_2 \text{ in } e_1 : t_1}$
(fix)		$\frac{A \cup \{x:t\} \vdash e : t}{A \vdash \text{fix } x : t. e : t}$

Fig. 1. Type System

Here c denotes constants (including integers, booleans and operators on these) and CT is a function mapping constants to their type. Furthermore we have variables, conditional, abstraction, application (denoted $@$), let-expressions and a fixed point operator.

The denotational semantics of the language is given in Fig. 2. We call this semantics the *standard semantics* in contrast to the semantics of the specializer for annotated *Exp*-programs, to be defined in Section 6.

In the semantics we assume a function CV mapping constant names to their meaning. Function $\lfloor \cdot \rfloor$ maps elements of a domain D into the corresponding lifted domain D_\perp and μ is the fixed point operator.

Semantic Domains:

$$V_{\text{bool}} = \{T, F\}_\perp$$

$$V_{\text{int}} = \mathbb{N}_\perp$$

$$V_{t_1 \rightarrow t_2} = V_{t_1} \rightarrow V_{t_2}$$

$$\mathcal{E}[c]\rho = \lfloor CV(c) \rfloor$$

$$\mathcal{E}[x]\rho = \rho(x)$$

$$\mathcal{E}[\text{if } e \text{ then } e' \text{ else } e'']\rho = \mathcal{E}[e]\rho \rightarrow \mathcal{E}[e']\rho \parallel \mathcal{E}[e'']\rho$$

$$\mathcal{E}[\lambda x:t.e]\rho = \lambda v \in V_t. \mathcal{E}[e]\rho[x \mapsto v]$$

$$\mathcal{E}[e@e']\rho = (\mathcal{E}[e]\rho)(\mathcal{E}[e']\rho)$$

$$\mathcal{E}[\text{let } x:t = e' \text{ in } e]\rho = \mathcal{E}[e]\rho[x \mapsto \mathcal{E}[e']\rho]$$

$$\mathcal{E}[\text{fix } x.e]\rho = \mu(\lambda v. \mathcal{E}[e]\rho[x \mapsto v])$$

Fig. 2. Denotational Semantics

5 Binding-Time Analysis

The aim of binding-time analysis is to *annotate* a source program with binding-time information about its individual parts. They are used to control the actions of a program specializer (see Section 6), but they can also be used during program design to convey static binding-time properties to a programmer. Binding-time analysis is the *automatic translation* of a source program to a language enriched with such annotations. In this section we describe the annotation-enriched language $2Exp$ and its syntactic properties. Section 6 contains its semantics. The algorithmic aspects will be covered elsewhere (see also [14]).

In the type inference framework, the annotation-enriched language is usually modeled by a *two-level* syntax [16, 7] having two versions of each language construct (with a non-underlined version corresponding to static/unfold and underlined corresponding to dynamic/residualize). We will follow this scheme, but to achieve polymorphism in unfold/residualize, we replace the underline notation by a superscript *annotation b* (binding-time value), which can also be a bound binding-time variable. Further we add *coercions (lifts)* and explicit binding-time abstraction and application, all of which are also considered annotations. This gives us the following syntax for annotated expressions $2Exp$:

$$\begin{aligned} e ::= & c \mid x \mid \text{if}^b e \text{ then } e \text{ else } e \mid \lambda^b x.e \mid e@^b e \mid \text{let } x = e \text{ in } e \mid \text{fix } x.e \\ & \Lambda\beta.e \mid e\{b\} \mid [\kappa \rightsquigarrow \kappa]e \end{aligned}$$

Note that there are no annotations on *let* and *fix*. Operationally this indicates that they can *always* be unfolded — this is semantically safe, since unfolding a *let*

or fix will not depend on dynamic (unavailable) data. It is, however, clear, that unfolding is not always desirable since it might lead to duplication or nonterminating specialization. In our problem set-up controlling unfolding is not part of the binding-time analysis proper, but is left to later analyses (possibly exploiting binding-time information). $\Lambda\beta.e$ is abstraction over binding-time variables, $\{\cdot\}$ denotes application of such abstractions to binding-time values and $[\kappa \rightsquigarrow \kappa']$ is a coercion from κ to κ' . The *binding times* b , *compound types* κ and type schemes σ are defined by:

$$\begin{aligned} b &::= S \mid D \mid \beta \\ \kappa &::= \kappa \rightarrow^b \kappa \mid \text{Int}^b \mid \text{Bool}^b \\ \sigma &::= \kappa \mid \forall \beta. \sigma \mid b \leq b' \Rightarrow \sigma \end{aligned}$$

Compound types are standard types with binding-time superscript annotations on every type and type constructor. E.g. $\text{Int}^D \rightarrow^S \text{Bool}^D$ is the type of a statically applicable (the value “S” is the binding-time of “ \rightarrow ”) predicate on dynamic integers giving dynamic results. We write t^b to match compound types κ .

The type scheme $\forall \beta. \sigma$ is the type of $\Lambda\beta.e$ expressing the parameterization with binding-time values. A *constraint* is a pair of two binding time values written $b_1 \leq b_2$. Constraints $S \leq S$, $S \leq D$ and $D \leq D$ are said to *hold*; $D \leq S$ does not hold. Then $e:b \leq b' \Rightarrow \sigma$ is read: if $b \leq b'$ holds then e has type σ . A binding-time judgement has a type environment A and a set of constraints C as assumptions. Thus the $b \leq b' \Rightarrow \sigma$ construct allows us to discharge an assumption and make it explicit in the type.

Our type system for inferring polymorphic binding-time information is given in Fig. 3. We assume a function \mathcal{CT} mapping constant names to their type (e.g. $\mathcal{CT}(5) = \text{Int}^S$ and $\mathcal{CT}(\text{not}) = \forall \beta. \text{Bool}^\beta \rightarrow^\beta \text{Bool}^\beta$).

Variables bound by let or fix can have types polymorphic in *binding-time variables* even though the standard type system is monomorphic. Thus the polymorphism in the analysis does not depend on the availability of polymorphism in the underlying standard type system (though it can, of course, accommodate such). Despite the adoption of a polymorphic binding-time typing rule for recursive definitions the binding-time type system is decidable as binding-time annotations are anchored to the standard types.

We illustrate polymorphism by an annotated Ackermann function:

```
let ack = fix a. $\Lambda\beta_i\beta_j\beta_{res}$ .
     $\lambda i.\lambda j.\text{if}^{\beta_i} i =^{\beta_i} 0 \text{ then } [\text{Int}^{\beta_j} \rightsquigarrow \text{Int}^{\beta_{res}}](j +^{\beta_j} 1)$ 
     $\text{else if}^{\beta_j} j =^{\beta_j} 0 \text{ then } [\text{Int}^{\beta_i} \rightsquigarrow \text{Int}^{\beta_{res}}](a\{\beta_i S \beta_i\} @ S_{(i - \beta_i)} @ S_1)$ 
     $\text{else } a\{\beta_i \beta_{res} \beta_{res}\}$ 
     $@ S_{(i - \beta_i)} @ S_{(a\{\beta_i \beta_j \beta_{res}\} @ S_i @ S_{(j - \beta_j)})}$ 
in ack{SSS}@S2@S3+ack{SDD}@S2@Sd+ack{DSD}@Sd@S3
```

Note how the function is used polymorphically both in external and recursive calls. The binding time description of the Ackermann function will be:

$$\forall \beta_i \beta_j \beta_{res}. \beta_i \leq \beta_{res} \Rightarrow \beta_j \leq \beta_{res} \Rightarrow \text{Int}^{\beta_i} \rightarrow \text{Int}^{\beta_j} \rightarrow \text{Int}^{\beta_{res}}$$

	(const) $A, C \vdash c : \mathcal{C}T(c)$
	(var) $A \bigcup \{x : \sigma\}, C \vdash x : \sigma$
(if)	$\frac{A, C \vdash e_1 : \text{Bool}^{b_1} \quad A, C \vdash e_2 : t^{b_2} \quad A, C \vdash e_3 : t^{b_2} \quad C \vdash b_1 \leq b_2}{A, C \vdash \text{if}^{b_1} e_1 \text{ then } e_2 \text{ else } e_3 : t^{b_2}}$
(abstr)	$\frac{A \bigcup \{x : t^b\}, C \vdash e_1 : t_1^{b_1} \quad C \vdash b_2 \leq b \quad C \vdash b_2 \leq b_1}{A, C \vdash \lambda^{b_2} x : t^b. e : t^b \rightarrow^{b_2} t_1^{b_1}}$
(appl)	$\frac{A, C \vdash e_1 : t_2^{b_2} \rightarrow^b t_1^{b_1} \quad A, C \vdash e_2 : t_2^{b_2} \quad C \vdash b \leq b_1 \quad C \vdash b \leq b_2}{A, C \vdash e_1 @^b e_2 : t_1^{b_1}}$
(let)	$\frac{A, C \vdash e_2 : \sigma \quad A \bigcup \{x : \sigma\}, C \vdash e_1 : \kappa}{A, C \vdash \text{let } x : \sigma = e_2 \text{ in } e_1 : \kappa}$
(fix)	$\frac{A \bigcup \{x : \sigma\}, C \vdash e : \sigma}{A, C \vdash \text{fix } x : \sigma. e : \sigma}$
(\forall -introduction)	$\frac{A, C \vdash e : \sigma}{A, C \vdash \forall \beta. e : \forall \beta. \sigma} \text{ (if } \beta \text{ not free in } A, C\text{)}$
(\forall -elimination)	$\frac{A, C \vdash e : \forall \beta. \sigma}{A, C \vdash e \{b\} : [b/\beta]\sigma}$
(\Rightarrow -introduction)	$\frac{A, C \bigcup \{b \leq b_1\} \vdash e : \sigma}{A, C \vdash e : b \leq b_1 \Rightarrow \sigma}$
(\Rightarrow -elimination)	$\frac{A, C \vdash e : b \leq b_1 \Rightarrow \sigma \quad C \vdash b \leq b_1}{A, C \vdash e : \sigma}$
(coerce)	$\frac{A, C \vdash e : \kappa \quad C \vdash \kappa \leq \kappa'}{A, C \vdash [\kappa \rightsquigarrow \kappa']e : \kappa'}$
(lookup-coerce)	$C \bigcup \{b \leq b'\} \vdash b \leq b'$
(id)	$C \vdash b \leq b$
(lift)	$C \vdash S \leq D$
(use)	$\frac{C \vdash b \leq b'}{C \vdash \text{Base}^b \leq \text{Base}^{b'}}$

Fig. 3. Polymorphic BTA

Definition 1. Every annotated expression e has an underlying standard expression denoted by $|e|$, which can be obtained by *erasing* annotations. We call $|e^{ann}|$ the *erasure* of e^{ann} . Conversely, e^{ann} is called a (*binding-time*) *completion* of $|e^{ann}|$. Similarly, $|\cdot|$ can be applied to type schemes and (binding time) environments. For expression $e \in \text{Exp}$ we write $A, C \vdash e : \sigma$ if e has a completion $e^{ann} \in 2\text{Exp}$ such that $A, C \vdash e^{ann} : \sigma$. \square

We now define the notion of *generic instance*. Our definition differs from the standard one since we have explicit polymorphism and coercions. We introduce the notation $C[]$ for a *coercion context* containing Λ 's, binding-time applications and coercions. Note that for any coercion context $C[]$ we have $|C[e]| = |e|$.

Definition 2. We say σ' is a *generic instance* of σ and write $\sigma \subseteq \sigma'$ if there exists a coercion context $C[]$ such that $x : \sigma \vdash C[x] : \sigma'$. \square

E.g. we have $(\forall \beta_3. \text{Int}^S \rightarrow^{\beta_3} \text{Int}^D) \subseteq (\forall \beta_1. \forall \beta_2. \beta_1 \leq \beta_2 \Rightarrow \forall \beta_3. \text{Int}^{\beta_1} \rightarrow^{\beta_3} \text{Int}^{\beta_2})$.

Definition 3. Given A, C we call σ_{pt} a *principal (binding-time) type* of $e \in \text{Exp}$ under A, C if

- $A, C \vdash e : \sigma_{pt}$
- $\forall \sigma : A, C \vdash e : \sigma \implies \sigma_{pt} \subseteq \sigma$

□

Our binding-time analysis has the *principality property* for binding-time types (corresponding to a principal typing property, but on binding time values); that is, *all* expressions have a principal type. The significance of the principal typing property for an expression $e \in \text{Exp}$ is that there is a single derivable binding-time type that can be used in a binding-time analysis for *all* possible contexts in which e might occur. This is the key to the modularity and partial completeness of the analysis with respect to unfolding of let- and fix-expressions. (See Section 8.)

Theorem 4. (Principal Typing Property)

Every expression $e \in \text{Exp}$ has a principal type under A, C if it has any (binding-time) type under A, C at all.

Proof. By induction over the standard type derivation tree (otherwise similar in style to [15]). □

For $A \vdash e : t$ it can be seen that e has a (principal) type for any completion A^{ann} of A if all type assumptions are first-order; that is, not of functional type.

A principal type of an expression is clearly not unique due to, amongst other things, reordering of \forall and $\cdot \leq \cdot \Rightarrow \cdot$. Note also that there are generally several different completions of an expression with the *same* (principal) type. This is due to the fact that principality only considers the *externally* visible binding-time behavior of an expression, but not its internal structure.

6 Specialization

In Fig. 4 we present the denotational semantics for binding-time analyzed programs. This is essentially a specializer consuming the information generated by the polymorphic binding-time analysis from the last section.

The semantic function takes two environments: a standard environment ρ , and a binding-time environment η mapping binding time variables to binding-time values in BtV . By abuse of notation we take $\eta(S) = S$ and $\eta(D) = D$. The specializer is “natural” in the sense that every clause for static expressions (including the rules for let and fix) is identical to the corresponding rule of the standard semantics and all rules for dynamic expressions simply evaluate subexpressions and emit code (corresponding to off-line partial evaluation). We assume a function $2CV$ mapping 2-level constant names to their meaning.

Note that there are now two kinds of values: real values that you can “do computations with” and residual values that are pure syntax to appear in the residual program. The functions *build-if*, *build-λ* and *build-@* take arguments

Semantic Domains:

$$\begin{aligned}
 2V_{\text{Bool}^s} &= \{T, F\}_{\perp} \\
 2V_{\text{Int}^s} &= \text{IN}_{\perp} \\
 BtV &= \{S, D\} \\
 2V_{\kappa_1 \rightarrow s_{\kappa_2}} &= 2V_{\kappa_1} \rightarrow 2V_{\kappa_2} \\
 2V_{t^D} &= \text{Exp} \\
 2V_{\forall \beta. \sigma} &= BtV \rightarrow (2V_{\sigma})_{\perp} \\
 2V_{b_1 \leq b_2 \Rightarrow \sigma} &= 2V_{\sigma}
 \end{aligned}$$

$$T[\![c]\!] \rho \eta = \lfloor 2CV(c) \rfloor$$

$$T[\![x]\!] \rho \eta = \rho(x)$$

$$\begin{aligned}
 T[\![\text{if } b \text{ e then } e' \text{ else } e'']\!] \rho \eta &= \text{case } \eta(b) \text{ of} \\
 &\quad S: T[\![e]\!] \rho \eta \rightarrow T[\![e']\!] \rho \eta \parallel T[\![e'']\!] \rho \eta \\
 &\quad D: \text{build-if}(T[\![e]\!] \rho \eta, T[\![e']\!] \rho \eta, T[\![e'']\!] \rho \eta)
 \end{aligned}$$

$$T[\![\lambda^b x : t^{b'} . e]\!] \rho \eta = \text{case } \eta(b) \text{ of}$$

$$\begin{aligned}
 &\quad S: \lambda v \in (V_{t^{b'}})_{\perp} . T[\![e]\!] \rho[x \mapsto v] \eta \\
 &\quad D: \text{build-}\lambda(x, t, T[\![e]\!] \rho \eta)
 \end{aligned}$$

$$T[\![e @^b e']\!] \rho \eta = \text{case } \eta(b) \text{ of}$$

$$\begin{aligned}
 &\quad S: (T[\![e]\!] \rho \eta)(T[\![e']\!] \rho \eta) \\
 &\quad D: \text{build-}@(T[\![e]\!] \rho \eta, T[\![e']\!] \rho \eta)
 \end{aligned}$$

$$T[\![\text{let } x : \sigma = e' \text{ in } e]\!] \rho \eta = T[\![e]\!] \rho[x \mapsto T[\![e']\!] \rho \eta] \eta$$

$$T[\![\text{fix } x. e]\!] \rho \eta = \mu(\lambda v. T[\![e']\!] \rho[x \mapsto v] \eta)$$

$$\begin{aligned}
 T[\![\text{Base}^{b_1 \rightsquigarrow b_2} e]\!] \rho \eta &= \text{case } (\eta(b_1), \eta(b_2)) \text{ of} \\
 &\quad (S, D): \text{build-const}(T[\![e]\!] \rho \eta) \\
 &\quad (S, S) \text{ or } (D, D): T[\![e]\!] \rho \eta \\
 &\quad (D, S): \text{error}
 \end{aligned}$$

$$T[\![\Lambda \beta. e]\!] \rho \eta = (\lambda b. T[\![e]\!] \rho \eta[\beta \mapsto b])$$

$$T[\![e \{ b \}]\!] \rho \eta = (T[\![e]\!] \rho \eta)(\eta(b))$$

Fig. 4. Denotational Semantics for Binding-time Analyzed Expressions

from $2V_{t^D}$ and return an element in $2V_{t^D}$ (all arguments and the result of appropriate matching types). Function *build-const* takes an argument from $2V_{\text{bool}^s}$ or $2V_{\text{int}^s}$ and returns an element in $2V_{\text{bool}^D}$, resp. $2V_{\text{int}^D}$.

7 Correctness

In Theorem 9 we state correctness. The definitions and proof is inspired by Gomard [7]. Correctness includes consistency of the specializer, *i.e.* for all closed $e \in \text{2Exp}$: if $\vdash e : \sigma$ then $T[\![e]\!] \rho \eta \in 2V_{\sigma}$ for all ρ, η . We write $\rho \in V_A$ iff for all x bound by ρ , $\rho(x) \in V_{A(x)}$.

Relation \mathcal{R} defines the relationship that (under suitable conditions) must hold between standard evaluation of an expression e and specialization of the same expression: if e is (first order) static, standard evaluation and specialization

should yield the same result; if e is (first order) dynamic, standard evaluation should yield the same result as standard evaluation of the specialized expression (we take $\forall \rho_d : \mathcal{E}[\perp]\rho_d = \perp$). Note that specialization might not terminate while standard evaluation does; this is expressed in the definition of \mathcal{R} using \sqsubseteq . The relation is extended to a logical relation on higher order and polymorphic types.

Definition 5. Given $A, C, 2CT, CV$ and $2CV$, the relation \mathcal{R} holds for $(e, \rho_s, \rho_d, \rho, \eta, \sigma) \in 2Exp \times 2VarEnv \times VarEnv \times VarEnv \times BtVarEnv \times Type$ iff

1. $A, C \vdash e : \sigma$ if $\rho_s \in V_A$.
2. One of the following holds
 - (a) $\sigma = \text{Base}^S$ and $T[e]\rho_s\eta = \mathcal{E}[|e|]\rho$
 - (b) $\sigma = \text{Base}^D$ or $\sigma = \kappa \rightarrow^D \kappa'$ and $\mathcal{E}[T[e]\rho_s\eta]\rho_d \sqsubseteq \mathcal{E}[|e|]\rho$
 - (c) $\sigma = \kappa' \rightarrow^S \kappa$ and $\forall e' : \mathcal{R}(e', \rho_s, \rho_d, \rho, \eta, \kappa')$ implies $\mathcal{R}(e @^S e', \rho_s, \rho_d, \rho, \eta, \kappa)$
 - (d) $\sigma = \forall \beta. \sigma'$ and $\forall b \in BtVal : \mathcal{R}(e\{b\}, \rho_s, \rho_d, \rho, \eta, \sigma'[b/\beta])$.
 - (e) $\sigma = b \leq b' \Rightarrow \sigma'$ and $C \vdash b \leq b'$ implies $\mathcal{R}(e, \rho_s, \rho_d, \rho, \eta, \sigma')$.

□

The specialization time environment ρ_s , the run time environment ρ_d and the standard evaluation environment ρ should *agree*: If a variable x is (first order) static, ρ_s and ρ hold the same value; if x is (first order) dynamic, ρ_s maps x to a new variable name which when looked up in ρ_d yields the same value as $\rho(x)$. This extends to higher order values in a natural way:

Definition 6. Given a set of identifiers, $VarSet$, three environments, ρ, ρ_s, ρ_d and a type environment A such that $\rho_s \in V_A$, we say that ρ_s, ρ_d, ρ *agree* on $VarSet$ iff $\forall x \in VarSet, \forall \eta \in BtVarEnv : \mathcal{R}(x, \rho_s, \rho_d, \rho, \eta, A(x))$. □

Definition 7. We say that $2CT, CV, 2CV$ *match* iff $\forall \rho_s, \rho_d, \rho, \eta : \mathcal{R}(c, \rho_s, \rho_d, \rho, \eta, 2CT(c))$. □

We say that a binding-time environment $\eta \in BtVarEnv$ is *ground* if it maps all binding-time variables to binding-time values S or D. We allow binding-time environments to be applicable to types and type environments in the obvious way (essentially working as substitutions).

Definition 8. Given a ground environment η and a coercion set C , the relation $\eta \models C$ (η solves C) is defined by:

$$\begin{aligned} \eta \models \{(b \leq b')\} &\text{ iff } (\eta(b), \eta(b')) \in \{(S, S), (S, D), (D, D)\} \\ \eta \models C_1 \cup C_2 &\text{ iff } \eta \models C_1 \text{ and } \eta \models C_2 \end{aligned}$$

□

Theorem 9. (Correctness)

$\forall e, \rho_s, \rho_d, \rho, A, C, \eta, \sigma, 2CT, CV, 2CV :$

$$\left. \begin{array}{l} 2CT, CV, 2CV \text{ match} \\ \eta \models C \\ \rho_s \in V_{\eta A} \\ \rho_s, \rho_d, \rho \text{ agree on } \text{FreeVars}(e) \\ A, C \vdash e : \sigma \end{array} \right\} \implies \mathcal{R}(e, \rho_s, \rho_d, \rho, \eta, \sigma)$$

Proof. Proceeds by induction over the derivation of $A, C \vdash e : \sigma$. The interesting cases are proved in Lemma 14 to 16 which can be found in appendix A. \square

8 Partial Completeness

We prove the subject expansion property for `let` and `fix`. This implies that performing binding-time analysis after unfolding `let`- or `fix`-expressions will not give any better binding-time types. Making different variants of a (top-level) function for each context in which it is used, corresponds to unfolding `let` and `fix` bound expressions. Thus our polymorphic binding time analysis yields as good results *without* changing the program at hand as any polyvariant binding time analysis using this technique. This is a *partial completeness* result for our analysis³ as it shows that the analysis is *invariant* under the equality induced by `let`- and `fix`-unfolding. With *induced* coercions it is also complete with respect to η -equality (not shown here). As can be expected it is *not* complete with respect to (non-linear) β -reduction and simplification of conditionals. Note, however, that the weaker subject *reduction* property holds also for these cases.

If e is an expression with k occurrences of variable x , we refer to the i 'th occurrence as $x^{(i)}$ and use $e[e'/x^{(i)}]$ to denote the substitution of e' for the i 'th occurrence of x .

The following lemma states that any type derivation tree can be cut at any point inserting variables with the same type as the cut branch. The converse holds too (that variables can be replaced by expressions of the same type). This is basically the well-known substitution lemma.

Lemma 10. *Let e be an expression with k occurrences of variable x . Then*

$$\begin{aligned} A, C \vdash e[e_i/x^{(i)}] : \sigma &\iff \\ (\exists \sigma_1 \dots \sigma_k) A, C \vdash e_i : \sigma_i \wedge A \bigcup \{x_1 : \sigma_1, \dots, x_k : \sigma_k\}, C \vdash e[x_i/x^{(i)}] : \sigma \end{aligned}$$

Proof. By induction over the type inference tree. \square

Theorem 11. (*Subject Expansion for let*)

$$A, C \vdash \text{let } x = e' \text{ in } e : \sigma \Leftrightarrow A, C \vdash e[e'/x] : \sigma.$$

Proof. (\Rightarrow) This is the subject-reduction property for `let`, which follows from Lemma 10. (\Leftarrow) Similar to Theorem 13 (see below). \square

The following lemma is used in the proof of subject expansion for `fix`.

Lemma 12. *If σ_{pt} is a principal type of fix $x.e \in \text{Exp}$ under A, C then σ_{pt} is also a principal type of e under $A \bigcup \{x : \sigma_{pt}\}, C$.*

³ It can't be both computable and totally complete for obvious reasons.

Proof. Assume it isn't; i.e., there exists σ with $\sigma \subseteq \sigma_{pt}$ and $\sigma_{pt} \not\subseteq \sigma$ such that $A \cup \{x : \sigma_{pt}\}, C \vdash e : \sigma$. From the definition of generic instantiation, Definition 2, it follows that $A \cup \{x : \sigma\}, C \vdash e : \sigma$. This implies $A, C \vdash \text{fix } x.e : \sigma$. Since σ_{pt} is principal for $\text{fix } x.e$ we get $\sigma_{pt} \subseteq \sigma$, which is in contradiction to our proof assumption. \square

Theorem 13. (*Subject Expansion for fix*)

$$A, C \vdash \text{fix } x.e : \sigma \Leftrightarrow A, C \vdash e[\text{fix } x.e/x] : \sigma.$$

Proof. (\Rightarrow) (Subject reduction) Follows from Lemma 10. (\Leftarrow) (Subject expansion) Let e contain k occurrences of x . Assume $A, C \vdash e[\text{fix } x.e/x] : \sigma$. By Lemma 10 there exist $\sigma_1, \dots, \sigma_k$ such that

1. $A, C \vdash \text{fix } x.e : \sigma_i$ for $1 \leq i \leq k$, and
2. $A \cup \{x_1 : \sigma_1, \dots, x_k : \sigma_k\}, C \vdash e[x_i/x^{(i)}] : \sigma$.

By Theorem 4 we know that $\text{fix } x.e$ has a principal type σ_{pt} under A, C . Thus $\sigma_{pt} \subseteq \sigma_i$ for all $1 \leq i \leq k$. From the definition of \subseteq , Definition 2, and point 2 it follows that

$$A \cup \{x : \sigma_{pt}\}, C \vdash e : \sigma.$$

Since, by Lemma 12, σ_{pt} is a principal type for e under $A \cup \{x : \sigma_{pt}\}, C$ we obtain $\sigma_{pt} \subseteq \sigma$. Now we can conclude from $A, C \vdash \text{fix } x.e : \sigma_{pt}$ together with $\sigma_{pt} \subseteq \sigma$ that $A, C \vdash \text{fix } x.e : \sigma$. \square

9 Implementation

A working prototype implementation of the system exists (described in [14]) using standard type inference methods. Some work is needed to improve the efficiency of the implementation. An important issue in the implementation is to keep the principal types small (to avoid big program size increases). Reductions capable of doing this are described in [14], but they are neither proved correct nor implemented efficiently.

References

1. A. Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17 (Selected papers of ESOP '90, the 3rd European Symposium on Programming)(1-3):3-34, Dec. 1991.
2. M. A. Bulyonkov. Extracting polyvariant binding time analysis from polyvariant specializer. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 59-65. ACM Press, 1993.
3. C. Consel. Polyvariant binding-time analysis for applicative languages. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 66-77. ACM Press, 1993.

4. C. Consel. A tour of Schism: A partial evaluation system for higher-order applicative languages. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 145–154. ACM Press, 1993.
5. C. Consel and P. Jouvelot. Separate polyvariant binding-time analysis. Research report, Pacific Software Research Center, Oregon Graduate Institute of Science and Technology, Beaverton, Oregon, USA, 1993.
6. M. Gengler and B. Rytz. A polyvariant binding time analysis. In *1992 ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 21–28. Yale University, 1992.
7. C. K. Gomard. A self-applicable partial evaluator for the lambda calculus: Correctness and pragmatics. *Transactions on Programming Languages and Systems*, 14(2):147–172, Apr. 1992.
8. S. Hunt and D. Sands. Binding time analysis: A new PERspective. In *Proc. ACM/IFIP Symp. on Partial Evaluation and Semantics Based Program Manipulation (PEPM), New Haven, Connecticut*, June 1991.
9. J. Jørgensen. Generating a compiler for a lazy language by partial evaluation. In *Proc. 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 258–268. ACM Press, Jan. 1992.
10. J. Launchbury. *Projection Factorisations in Partial Evaluation*. PhD thesis, University of Glasgow, Jan. 1990.
11. T. Mogensen. Binding time analysis for polymorphically typed higher order languages. In J. Diaz and F. Orejas, editors, *Proc. Int. Conf. Theory and Practice of Software Development (TAPSOFT)*, pages 298–312, March 1989. LNCS 352.
12. C. Mossin. Similix binding time debugger manual, system version 4.0. Included in Similix distribution, Sept. 1991.
13. C. Mossin. Partial evaluation of general parsers. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 13–21. ACM Press, 1993.
14. C. Mossin. Polymorphic binding time analysis. Master’s thesis, DIKU, University of Copenhagen, Denmark, July 1993.
15. A. Mycroft. Polymorphic type schemes and recursive definitions. In *Proc. 6th Int. Conf. on Programming, LNCS 167*, 1984.
16. H. Nielson and F. Nielson. Automatic binding time analysis for a typed lambda calculus. *Science of Computer Programming*, 10:139–176, 1988.

A Proof of Correctness

We prove Theorem 9. Let $\mathcal{H}(A, C \vdash e : \sigma)$ be the induction hypothesis. Though Gomard’s correctness theorem has been slightly altered *w.r.t.* to standard types, coercion sets and binding time variables, the induction cases from his proof [7] carry directly over to our proof. We show the cases for fix, binding-time abstraction and binding time application. The let case is similar to the fix case, constrained binding time types are trivial and the remaining cases correspond to cases by Gomard.

Note that without losing generality any type scheme can be written in the form $\forall \beta_1 \dots \forall \beta_n. b_1' \leq b_1'' \Rightarrow \dots b_m' \leq b_m'' \Rightarrow \kappa$. Let $\mathcal{CT}, \mathcal{CV}$ and \mathcal{CF} be given throughout the proof.

Lemma 14. $\forall e, e' : \mathcal{H}(A \cup \{x : \sigma\}, C \vdash e : \sigma) \Rightarrow \mathcal{H}(A, C \vdash \text{fix } x.e : \sigma)$

Proof. Let ρ_s, ρ_d, ρ and η be given such that the conditions of Theorem 9 hold. Define $\rho'_s = \rho_s[x \mapsto val]$ and $\rho' = \rho[x \mapsto val']$, where val, val' are given such that $\mathcal{R}(x, \rho'_s, \rho_d, \rho', \eta, \sigma)$. From the induction hypothesis we have

$$\mathcal{R}(e, \rho'_s, \rho_d, \rho', \eta, \sigma) \quad (1)$$

The idea is to use (1) to prove the induction step of an induction proof. We have that $\eta\sigma$ has the form $\forall\beta_1 \dots \forall\beta_n. b_1' \leq b_m'' \Rightarrow \dots b_m' \leq b_m'' \Rightarrow \kappa_1 \rightarrow^S \dots \rightarrow^S \kappa_p$ where κ_p is either Base^S or one of Base^D or $\kappa' \rightarrow^D \kappa''$.

We only prove the case where $\kappa_p = \text{Base}^D$. Case $\kappa_p = \kappa' \rightarrow \kappa''$ is identical and $\kappa_p = \text{Base}^S$ similar though simpler.

Let $b_1 \dots b_n$ and $e_1 \dots e_{p-1}$ be given such that $\mathcal{R}(e_i, \rho_s, \rho_d, \rho, \eta, \kappa_i)$. Now $\mathcal{R}(\text{fix } x.e, \rho_s, \rho_d, \rho, \eta, \eta\sigma)$ may also be written

$$\mathcal{R}((\text{fix } x.e)\{b_1\} \dots \{b_n\} @^S_{e_1} @^S \dots @^S_{e_{p-1}}, \rho_s, \rho_d, \rho, \eta, \kappa_p)$$

This is equivalent to

$$\begin{aligned} & \mathcal{E}[\![T((\text{fix } x.e)\{b_1\} \dots \{b_n\} @^S_{e_1} @^S \dots @^S_{e_{p-1}}]]\!] \rho_s \eta] \rho_d \sqsubseteq \\ & \mathcal{E}[\![(\text{fix } x.e)\{b_1\} \dots \{b_n\} @^S_{e_1} @^S \dots @^S_{e_{p-1}}]\!] \rho \end{aligned}$$

This can be rewritten to

$$\begin{aligned} & \mathcal{E}[\!(T[\![\text{fix } x.e]\!] \rho_s \eta) b_1 \dots b_n (T[\![e_1]\!] \rho_s \eta) \dots (T[\![e_{p-1}]\!] \rho_s \eta)]\!] \rho_d \sqsubseteq \\ & (\mathcal{E}[\![\text{fix } x.e]\!] \rho)(\mathcal{E}[\![e_1]\!] \rho) \dots (\mathcal{E}[\![e_{p-1}]\!] \rho) \end{aligned}$$

by the semantics we get

$$\begin{aligned} & \mathcal{E}[\![\mu(\lambda v. T[\![e]\!] \rho_s[x \mapsto v]) b_1 \dots b_n (T[\![e_1]\!] \rho_s \eta) \dots (T[\![e_{p-1}]\!] \rho_s \eta)]\!] \rho_d \sqsubseteq \\ & \mu(\lambda v. \mathcal{E}[\![e]\!] \rho[x \mapsto v])(\mathcal{E}[\![e_1]\!] \rho) \dots (\mathcal{E}[\![e_{p-1}]\!] \rho) \end{aligned}$$

We prove this by simultaneous fixed point induction. If we use \perp_{std} to denote $(\lambda v_1 \dots \lambda v_{p-1}. \perp)$ and \perp_d to denote $(\Lambda b v_1 \dots \Lambda b v_n. \lambda v_1 \dots \lambda v_{p-1}. \perp)$ we can write the induction base as

$$\mathcal{E}[\!\perp_d b_1 \dots b_n (T[\![e_1]\!] \rho_s \eta) \dots (T[\![e_{p-1}]\!] \rho_s \eta)]\!] \rho_d \sqsubseteq \perp_{std}(\mathcal{E}[\![e_1]\!] \rho) \dots (\mathcal{E}[\![e_{p-1}]\!] \rho)$$

If we use f to denote $(\lambda v. \mathcal{E}[\![e]\!] \rho[x \mapsto v])$ and f_s to denote $(\lambda v. T[\![e]\!] \rho_s[x \mapsto v]\eta)$ then to prove the induction step, we prove for all m :

$$\begin{aligned} & \mathcal{E}[\!(f_s^m \perp_d) b_1 \dots b_n (T[\![e_1]\!] \rho_s \eta) \dots (T[\![e_{p-1}]\!] \rho_s \eta)]\!] \rho_d \sqsubseteq (f_s^m \perp_{std})(\mathcal{E}[\![e_1]\!] \rho) \dots (\mathcal{E}[\![e_{p-1}]\!] \rho) \\ & \implies \mathcal{E}[\!(f_s^{m+1} \perp_d) b_1 \dots b_n (T[\![e_1]\!] \rho_s \eta) \dots (T[\![e_{p-1}]\!] \rho_s \eta)]\!] \rho_d \\ & \sqsubseteq (f_s^{m+1} \perp_{std})(\mathcal{E}[\![e_1]\!] \rho) \dots (\mathcal{E}[\![e_{p-1}]\!] \rho) \end{aligned} \quad (2)$$

This follows from (where we exploit the fact, that x does not appear free in e_i)

$$\begin{aligned} & \mathcal{E}[\!(f_s^{m+1} \perp_d) b_1 \dots b_n (T[\![e_1]\!] \rho_s \eta) \dots (T[\![e_{p-1}]\!] \rho_s \eta)]\!] \rho_d = \\ & \mathcal{E}[\!(\lambda v. T[\![e]\!] \rho_s[x \mapsto v]\eta) (f_s^m \perp_d) b_1 \dots b_n (T[\![e_1]\!] \rho_s \eta) \dots (T[\![e_{p-1}]\!] \rho_s \eta)]\!] \rho_d = \\ & \mathcal{E}[\!(T[\![e]\!] \rho_s[x \mapsto (f_s^m \perp_d)]\eta) b_1 \dots b_n (T[\![e_1]\!] \rho_s \eta) \dots (T[\![e_{p-1}]\!] \rho_s \eta)]\!] \rho_d = \\ & \mathcal{E}[\![T[\![e\{b_1\} \dots \{b_n\}]\!] @^S_{e_1} @^S \dots @^S_{e_{p-1}}]]\!] \rho_s[x \mapsto (f_s^m \perp_d)]\eta] \rho_d \end{aligned} \quad (3)$$

If we in the definition of ρ_s' and ρ let val assume the value $f_s^m \perp_d$ and val' assume the value $f^m \perp_{std}$, then the (local) induction hypothesis gives us $\mathcal{R}(x, \rho_s', \rho_d, \rho', \eta, \sigma)$ and thus by (1) we have $\mathcal{R}(e, \rho_s', \rho_d, \rho', \eta, \sigma)$. This allows us to rewrite (3) to

$$\begin{aligned} & \subseteq \mathcal{E}[\![|e\{b_1\} \dots \{b_n\} @ S_{e_1} @ S \dots @ S_{e_{p-1}}|\!] \rho[x \mapsto (f_s^m \perp_{std})] = \\ & (\mathcal{E}[\![|e|\!] \rho[x \mapsto (f^m \perp_{std})])(\mathcal{E}[\![|e_1|\!] \rho) \dots (\mathcal{E}[\![|e_{p-1}|\!] \rho) = \\ & (\lambda v. \mathcal{E}[\![|e|\!] \rho[x \mapsto v])(f^m \perp_{std})(\mathcal{E}[\![|e_1|\!] \rho) \dots (\mathcal{E}[\![|e_{p-1}|\!] \rho) = \\ & (f^{m+1} \perp_{std})(\mathcal{E}[\![|e_1|\!] \rho) \dots (\mathcal{E}[\![|e_{p-1}|\!] \rho) \end{aligned}$$

□

Lemma 15. $\forall e : \mathcal{H}(A, C \vdash e : \sigma) \Rightarrow \mathcal{H}(A, C \vdash \Lambda\beta.e : \forall\beta.\sigma)$

Proof. Let ρ_s, ρ_d, ρ and η be given such that the conditions of Theorem 9 hold. Let $b \in BtVal$ be given and let $\eta' = \eta[\beta \mapsto b]$. Since $\text{FreeVars}(e) = \text{FreeVars}(\Lambda\beta.e)$, we have by the induction hypothesis that $\mathcal{R}(e, \rho_s, \rho_d, \rho, \eta', \eta'\sigma)$ holds.

We have that $\eta'\sigma$ has the form $\forall\beta_1 \dots \forall\beta_n. b_1' \leq b_1'' \Rightarrow \dots b_m' \leq b_m'' \Rightarrow \kappa_1 \rightarrow S \dots \rightarrow S_{\kappa_{p+1}}$ where κ_{p+1} is either Base^S or one of Base^D or $\kappa' \rightarrow^D \kappa''$. Let $b_1 \dots b_n \in BtVal$ and $e_1 \dots e_{p-1}$ be given such that $\mathcal{R}(e_i, \rho_s, \rho_d, \rho, \eta', \kappa_i)$. Now we have

$$\mathcal{R}(e\{b_1\} \dots \{b_n\} @ S_{e_1} @ S \dots @ S_{e_{p-1}}, \rho_s, \rho_d, \rho, \eta', \kappa_{p+1})$$

Assume $\kappa_{p+1} = \text{Base}^S$ ($\kappa_{p+1} = \text{Base}^D$ or $\kappa_{p+1} = \kappa' \rightarrow^D \kappa''$ are similar). Then

$$\mathcal{E}[\![|e\{b_1\} \dots \{b_n\} @ S_{e_1} @ S \dots @ S_{e_p}|\!] \rho = T[\![|e\{b_1\} \dots \{b_n\} @ S_{e_1} @ S \dots @ S_{e_p}|\!] \rho_s \eta'$$

This can be rewritten as:

$$(\mathcal{E}[\![|e|\!] \rho)(\mathcal{E}[\![|e_1|\!] \rho) \dots (\mathcal{E}[\![|e_p|\!] \rho) = (T[\![|e|\!] \rho_s \eta') b_1 \dots b_n (T[\![|e_1|\!] \rho_s \eta') \dots (T[\![|e_p|\!] \rho_s \eta')$$

By alpha-conversion can assume that β does not appear free in e_i :

$$(\mathcal{E}[\![|e|\!] \rho)(\mathcal{E}[\![|e_1|\!] \rho) \dots (\mathcal{E}[\![|e_p|\!] \rho) = (T[\![|e|\!] \rho_s \eta') b_1 \dots b_n (T[\![|e_1|\!] \rho_s \eta) \dots (T[\![|e_p|\!] \rho_s \eta)$$

Now by using the binding time application rule and the definition of η

$$\begin{aligned} & (\mathcal{E}[\![|(\Lambda\beta.e)\{b\}|\!] \rho)(\mathcal{E}[\![|e_1|\!] \rho) \dots (\mathcal{E}[\![|e_p|\!] \rho) = \\ & (T[\![|(\Lambda\beta.e)\{b\}|\!] \rho_s \eta) b_1 \dots b_n (T[\![|e_1|\!] \rho_s \eta) \dots (T[\![|e_p|\!] \rho_s \eta) \end{aligned}$$

Apply the binding time application rule and the @-rule to get

$$\begin{aligned} & \mathcal{E}[\![|\Lambda\beta.e\{b\} \dots \{b_n\} @ S_{e_1} @ S \dots @ S_{e_p}|\!] \rho = \\ & T[\![|\Lambda\beta.e\{b\} \dots \{b_n\} @ S_{e_1} @ S \dots @ S_{e_p}|\!] \rho_s \eta \end{aligned}$$

This implies $\mathcal{R}(\Lambda\beta.e\{b\}, \rho_s, \rho_d, \rho, \eta, \eta([b/\beta]\sigma))$, and the desired conclusion $\mathcal{R}(\Lambda\beta.e, \rho_s, \rho_d, \rho, \eta, \eta(\forall\beta.\sigma))$ follows from the definition of \mathcal{R} . □

Lemma 16. $\forall e, b : \mathcal{H}(A, C \vdash e : \forall\beta.\sigma) \Rightarrow \mathcal{H}(A, C \vdash e\{b\} : [b/\beta]\sigma)$

Proof. Let ρ_s, ρ_d, ρ and η be given such that the conditions of Theorem 9 hold. From $\text{FreeVars}(e) = \text{FreeVars}(e\{b\})$, we have $\mathcal{R}(e, \rho_s, \rho_d, \rho, \eta, \eta(\forall\beta.\sigma))$ by the induction hypothesis. The conclusion $\mathcal{R}(e\{b\}, \rho_s, \rho_d, \rho, \eta, \eta([b/\beta]\sigma))$ follows immediately. □

Shapely Types and Shape Polymorphism

C. Barry Jay¹ and J.R.B. Cockett²

¹ School of Computing Sciences, University of Technology, Sydney, PO Box 123 Broadway, 2007, Australia

² Department of Computer Science, University of Calgary, Calgary, Alberta, T2N 1N4, Canada

Abstract. Shapely types separate data, represented by lists, from shape, or structure. This separation supports shape polymorphism, where operations are defined for arbitrary shapes, and shapely operations, for which the shape of the result is determined by that of the input, permitting static shape checking. They include both arrays and the usual algebraic types (of trees, graphs, etc.), and are closed under the formation of initial algebras.

1 Introduction

Consider the operation `map` which applies a function to each element of a list. In existing functional languages, its type is

$$(\alpha \rightarrow \beta) \rightarrow \alpha \text{ list} \rightarrow \beta \text{ list}$$

where α and β may range over any types. This *data polymorphism* allows the data (α and β) to vary, but uses a fixed shape, `list`. *Shape polymorphism* fixes the data, but allows the shape to vary, so that, for types A and B , instances of `map` include

$$(A \rightarrow B) \rightarrow A \text{ tree} \rightarrow B \text{ tree}$$

and

$$(A \rightarrow B) \rightarrow A \text{ matrix} \rightarrow B \text{ matrix}$$

In each case `map(f)` applies f to the data (the leaves or entries), while leaving the shape fixed. Typically, both kinds of polymorphism co-exist, so that `map` can vary both its data and its shape.

Shape polymorphism applies to the usual algebraic types, (and others, such as matrices) but, being restricted to covariant constructions, does not address contravariant types of, say, functions.

The appropriate class of types are the *shapely types*, whose shape and data can be separated. With only one kind of data, this separation is represented by a pullback

$$\begin{array}{ccc} FA & \xrightarrow{\text{data}} & A \text{ list} \\ \downarrow \text{shape} & \lrcorner & \downarrow \text{length} \\ \text{Shapes} & \xrightarrow{\text{arity}} & N \end{array}$$

Values of a shapely type FA are uniquely determined by a list of A 's and a shape of type Shapes , such that the arity of the shape equals the length of the list. Shape polymorphism arises when the operations on the data and shape are completely independent of each other.

It often happens, particularly in scientific computation and data-processing, that the shape can influence the data, but not conversely. If computation is restricted to such *shapely operations* then the shape information can be treated as if it is part of the type system, with all shape computations, including shape checking, performed before executing the list operations. The latter can then be optimised, or run in parallel, without sacrificing (and indeed strengthening) the benefits of typing.

These ideas can be understood within a semantics based on sets, or of bottomless c.p.o's, but for generality (and flexibility) are presented in a locos [Coc90], which provides a minimal setting for working with both lists and pullbacks. In particular, no higher types are assumed (as is shown feasible in the language Charity[CF92]) since they contribute nothing to the theory of shape. A calculus of shapely functors and natural transformations is introduced, with the shapely type constructors being those functors which have a shapely transformation to the list functor (or a multi-parameter analogue of it).

Most of this paper is devoted to the introduction of concepts. The main result of the paper is that the shapely type constructors are closed under the construction of initial algebras. That is, once we have lists then we have all the other “algebraic” types of trees, graphs, queues, etc. The additional presence of arrays, and other types defined by pullbacks, means that the shapely types lie between these algebraic types, and the class of initial algebras for arbitrary functors. Full details of proofs can be found in [JC94].

2 Locoses

The types and operations are modelled by the objects and arrows of a category \mathcal{C} . It must have lists (and the underlying products and coproducts required to define them) and enough pullbacks to work with shapes. Specifying such a class of pullbacks (as was done for the Boolean categories of [Man92]) at this stage would impose an unwelcome burden so, to simplify slightly, we will assume that we have all pullbacks, and work in a finitely complete, distributive category (or extensive category [CLW93]) which has all list objects, i.e. a *locos* [Coc90]. Being extensive is equivalent to requiring that all coproduct diagrams have disjoint (monomorphic) inclusions, and are stable under pulling back. Examples include the usual semantic categories, including those of sets, bottomless c.p.o.'s, or even topological spaces, any one of which will suffice to illustrate the ideas below.

Let us fix some notation. If $f : C \rightarrow A$ and $g : C \rightarrow B$ are morphisms then $\langle f, g \rangle : C \rightarrow A \times B$ is their pairing. The left and right projections from the product are $\pi_{A,B}$ and $\pi'_{A,B}$ (respectively) and the unique morphism to the terminal object is $!_A : A \rightarrow 1$. The symmetry for the product is denoted $c_{A,B} : A \times B \rightarrow B \times A$. Dually, the coproduct inclusions are given by $\iota_{A,B} : A \rightarrow A + B$ and $\iota'_{A,B} : B \rightarrow A +$

B. If $f : A \rightarrow C$ and $g : B \rightarrow C$ then their case analysis is given by $[f, g] : A + B \rightarrow C$. The functors $\Pi, \Sigma : \mathcal{C}^n \rightarrow \mathcal{C}$ denote chosen n -fold products and coproducts, respectively, and $\Delta : \mathcal{C} \rightarrow \mathcal{C}^n$ is the diagonal functor.

The distributive law is witnessed by a natural isomorphism

$$d_{A,B,C} : A \times (B + C) \rightarrow (A \times B) + (A \times C)$$

whose inverse is $[\text{id} \times \iota, \text{id} \times \iota']$.

Subscripts on natural transformations will be omitted unless required to disambiguate an expression.

A *pullback* is a commuting square

$$\begin{array}{ccc} P & \xrightarrow{q} & B \\ \downarrow p \perp & & \downarrow g \\ A & \xrightarrow{f} & C \end{array}$$

such that, for every pair of morphism $x : X \rightarrow A$ and $y : X \rightarrow B$ such that $f \circ x = g \circ y$ there is a unique morphism $z : X \rightarrow P$ such that $p \circ z = x$ and $q \circ z = y$. We denote z by $\langle x, y \rangle$ (the usual pairing is a special case).

The list constructor is a functor $L : \mathcal{C} \rightarrow \mathcal{C}$. Its basic operations are denoted

$$\text{nil} : 1 \rightarrow LA \tag{1}$$

$$\text{cons} : A \times LA \rightarrow LA \tag{2}$$

$$\text{foldr}(x, h) : LA \rightarrow C \tag{3}$$

where $x : B \rightarrow C$ and $h : A \times C \rightarrow C$ are morphisms. **nil** and **cons** are the usual constructors, while **foldr**(x, h) is the unique morphism making the following diagram commute

$$\begin{array}{ccccc} B & \xrightarrow{\langle \text{nil}, \text{id} \rangle} & LA \times B & \xleftarrow{\text{cons} \times \text{id}} & A \times LA \times B \\ & \searrow x & \downarrow \text{foldr}(x, h) & & \downarrow \text{id} \times \text{foldr}(x, h) \\ & & C & \xleftarrow{h} & A \times C \end{array}$$

It follows that $[\text{nil}, \text{cons}] : 1 + (A \times LA) \rightarrow LA$ is an isomorphism, which expresses LA as a coproduct.

From these primitives we can construct the usual family of list operations, whose notation is a mixture of the list notation of [BW88] and categorical notation for monads:

$Lf : LA \rightarrow LB$	is <code>map(f)</code> for $f : A \rightarrow B$
$\eta : A \rightarrow LA$	makes singleton lists
$@ : LA \times LA \rightarrow LA$	is append
$snoc : LA \times A \rightarrow LA$	is <code>cons</code> on the tail of the list
$\mu : L^2 A \rightarrow LA$	flattens a list of lists
$g^* : LA \rightarrow LB$	is $\mu \circ Lg$ for $g : A \rightarrow LB$.

Many elementary results about lists in locoses can be found in [Jay93b].

$L1$ is a natural numbers object N with zero 0 and successor S given by `nil` and `cons` respectively. Then $\eta = \text{one}$ and $@ = +$ is addition and $\mu : LN \rightarrow N$ is summation. Let `Eq` be the equality on N . The *length* of a list object LA is $L! = \# : LA \rightarrow N$.

Define `shunt` : $LA \times LA \rightarrow LA \times LA$ to be

$$LA \times LA \cong LA + (LA \times A \times LA) \xrightarrow{[(\text{id}, \text{nil}), \text{snoc} \times \text{id}]} LA \times LA$$

where the isomorphism is given by the coproduct decomposition of LA and the distributive law. Then

$$\text{split} : N \times LA \rightarrow LA \times LA$$

is given by `foldr((nil, id), shunt)`. It divides a list into two segments, whose first, initial segment, has length given by the first projection (if the list is long enough). Define

$$\text{take} = \pi \circ \text{split} : N \times LA \rightarrow LA \quad (4)$$

$$\text{drop} = \pi' \circ \text{split} : N \times LA \rightarrow LA \quad (5)$$

Lemma 1. $\text{@} \circ \text{split} = \pi'$. Hence, we have a pullback

$$\begin{array}{ccc} LC \times LC & \xrightarrow{\langle \#, \pi, @ \rangle} & N \times LC \\ \downarrow & \lrcorner & \downarrow \\ 1 & \xrightarrow{\text{true}} & \text{bool} \end{array}$$

$$\text{Eq} \circ (\pi, \# \circ \text{take})$$

Proof. Both sides of the equation equal `foldr(id, id)`.

Given $x : X \rightarrow N \times LC$ for which $\text{Eq} \circ (\pi, \text{take}) \circ x = \text{true}$ then the induced morphism into the pullback is $\text{split} \circ x$. \square

The powers \mathcal{C}^n of \mathcal{C} are also locoses, and they have a right \mathcal{C} -action. That is, a functor $\mathcal{C}^n \times \mathcal{C} \rightarrow \mathcal{C}^n$ which maps $A = (A_1, A_2, \dots, A_n)$ and B to

$$A \times B = (A_1 \times B, A_2 \times B, \dots, A_n \times B)$$

and has the obvious action on morphisms.

3 Shapely Functors and Transformations

The essential ideas of this section can be found in [Jay93a].

A *strength* for a functor $F : \mathcal{C}^m \rightarrow \mathcal{C}$ is a natural transformation

$$\tau_{A,B} : FA \times B \rightarrow F(A \times B)$$

which satisfies the usual associativity and unicity axioms. More generally, a strength for $F : \mathcal{C}^m \rightarrow \mathcal{C}^n$ is given by a strength for each of its projections onto \mathcal{C} . See [CS92] for an account of the connections between strength and fibrations.

(F, τ) is a *shapely functor* if F is *stable* (preserves all pullbacks) and τ is a strength for it. Then $F1$ is the *object of F -shapes* and $\# = F! : FA \rightarrow F1$ is the *shape* of FA (generalising the length of a list). As F is stable, the following diagram is a pullback

$$\begin{array}{ccc} F(A \times B) & \xrightarrow{F\pi'} & FB \\ F\pi \downarrow & \lrcorner & \downarrow \# \\ FA & \xrightarrow{\#} & F1 \end{array}$$

If $FA \times_{\#} FB$ is a canonical choice of pullback (representing pairs that have the same shape) then there is an isomorphism

$$\text{zip} : FA \times_{\#} FB \rightarrow F(A \times B)$$

which is inverse to $\langle F\pi, F\pi' \rangle$ and generalises the usual `zip` of lists.

The shapely functors include the product, coproduct and list functors, and are closed under composition and pairing.

A natural transformation $\alpha : F \Rightarrow G$ is *cartesian* if, for every morphism $f : A \rightarrow B$, the following square is a pullback

$$\begin{array}{ccc} FA & \xrightarrow{\alpha_A} & GA \\ Ff \downarrow & \lrcorner & \downarrow Gf \\ FB & \xrightarrow{\alpha_B} & GB \end{array}$$

A *strong* natural transformation $(F, \tau_1) \Rightarrow (G, \tau_2)$ between strong functors is a natural transformation $\alpha : F \Rightarrow G$ that commutes with the strengths, i.e.

$$\begin{array}{ccc} FA \times B & \xrightarrow{\alpha_A \times \text{id}} & GA \times B \\ \tau_1 \downarrow & & \downarrow \tau_2 \\ F(A \times B) & \xrightarrow{\alpha_{A \times B}} & G(A \times B) \end{array}$$

If, further, F and G are shapely and α is cartesian, then α is a *shapely transformation* and F is *shapely over* G by α .

A consequence of being an extensive category is that the coproduct inclusions are shapely; cartesian-ness is by definition, and the strength is given by the distributive law.

Projections from the product $\times : \mathcal{C}^2 \rightarrow \mathcal{C}$, though strong, are never shapely. (If it were so then all the transformations of interest would be shapely.) Instead, given $f : A \rightarrow C$ and $g : B \rightarrow D$ we have the pullback

$$\begin{array}{ccc} A \times D & \xrightarrow{\pi} & A \\ f \times \text{id} \downarrow & \lrcorner & \downarrow f \\ C \times D & \xrightarrow{\pi} & C \end{array}$$

which shows that the transformation $\pi : (-) \times D \Rightarrow \text{id} : \mathcal{C} \rightarrow \mathcal{C}$ is cartesian. Hence, $\pi_{A,B}$ is shapely in A .

The shapely transformations are closed under vertical and horizontal composition, so that for each locos \mathcal{C} we have a 2-category of shapely functors and natural transformations. They are also closed under pairing and case analysis. Here are two more general results.

Proposition 2. *Let $F : \mathcal{C}^2 \rightarrow \mathcal{C}$ be a shapely functor and let $G, H : \mathcal{C} \rightarrow \mathcal{C}$ be any functors. Suppose that for each object B the transformation $\alpha_{A,B} : F(A, B) \rightarrow GA$ is cartesian in A . Similarly, suppose that for each object A the transformation $\beta_{A,B} : F(A, B) \rightarrow HB$ is cartesian in B . Then*

$$\langle \alpha_{A,B}, \beta_{A,B} \rangle : F(A, B) \rightarrow GA \times HB$$

is a cartesian in both A and B .

Proof. Consider a commuting square

$$\begin{array}{ccc} X & \xrightarrow{\langle x, y \rangle} & GA \times HB \\ z \downarrow & & \downarrow Gg \times Hh \\ F(A', B') & \xrightarrow{\langle \alpha, \beta \rangle} & GA' \times HB' \end{array}$$

Then x and z induce a unique morphism $x' : X \rightarrow F(A, B')$ by the cartesian-ness of α . Similarly y and z induce a morphism $y' : X \rightarrow F(A', B)$. As F is shapely, x' and y' induce the desired morphism $X \rightarrow F(A, B)$. \square

Theorem 3. If $\alpha : F \Rightarrow G$ and $\beta : H \times G \Rightarrow G$ are shapely transformations, then

$$\gamma_A = \text{foldr}(\alpha_A, \beta_A) : LHA \times FA \rightarrow GA$$

is a shapely transformation.

Proof. The result generalises [Jay93a, Theorem 2.6] without changing the proof. \square

Corollary 4. $@ = \text{foldr}(\text{nil}, \text{cons})$ and $\mu = \text{foldr}(\text{nil}, @)$ are shapely. \square

The following lemma shows how strength and cartesian-ness interact.

Lemma 5. If $\alpha : F \Rightarrow G$ is cartesian and (G, τ_2) is shapely then there is a unique strength τ_1 for F such that (F, τ_1) and α are both shapely.

Proof. If F has a strength τ_1 that makes α strong then the following diagram must commute.

$$\begin{array}{ccc}
FA \times B & \xrightarrow{\alpha \times \text{id}} & GA \times B \\
& \searrow \tau_1 & \swarrow \tau_2 \\
& F(A \times B) & G(A \times B) \\
\pi \downarrow & \nearrow F\pi & \downarrow G\pi \\
FA & \xrightarrow{\alpha} & GA
\end{array}$$

Since the square is a pullback, this determines τ_1 uniquely. Conversely, this pullback can be used to define τ_1 . Its naturality, associativity and unicity are all inherited from that of τ_2 . \square

4 Shapely Types

A functor $F : \mathcal{C} \rightarrow \mathcal{C}$ is a shapely type constructor if it is shapely over L . More generally, a functor $F : \mathcal{C}^n \rightarrow \mathcal{C}^n$ is a *shapely type constructor* if it is shapely over $\Delta \Pi L^n$. Equivalently, each of the projections of F onto \mathcal{C} must be shapely over ΠL^n . If δ is the relevant shapely transformation then we have the pullback

$$\begin{array}{ccc}
F(A_i) & \xrightarrow{\delta} & \Pi(LA_i) \\
\# \downarrow & \lrcorner & \downarrow \# \\
F1 & \xrightarrow{\delta_1} & N^n
\end{array}$$

$F(A_i)$ is a (tuple of) shapely types, and δ_1 is also known as the *arity* of $F1$.

The shapes can be thought of as having fixed numbers of holes or entries of each type, which are filled in by the data.

Of course, lists are shapely by the identity transformation. Also, $\mu : L^2 \Rightarrow L$ makes L^2 a shapely type constructor, with shapes given by lists of numbers.

The shape of a matrix is given by its dimensions, which are of type N^2 . The defining pullback is

$$\begin{array}{ccc} MA & \longrightarrow & LA \\ \downarrow & \lrcorner & \downarrow \# \\ N^2 & \xrightarrow{*} & N \end{array}$$

where MA is the type of matrices with entries from A . (They can also be defined as vectors of vectors [Jay93a]. Matrix multiplication, and general operations of linear algebra can all be defined in this way.)

Binary trees with leaves labelled by A 's have shapes given by (unlabelled) trees, whose arity is their number of leaves, and data given by their list of leaves.

$$\begin{array}{ccc} TA & \xrightarrow{\text{leaves}} & LA \\ \# \downarrow & \lrcorner & \downarrow \# \\ T1 & \xrightarrow{\text{leaves}} & N \end{array}$$

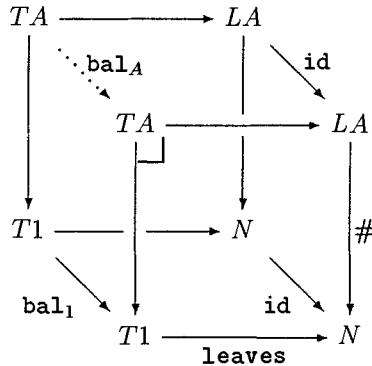
The pullback construction does not determine the order of elements in the list of leaves, which could be listed from left to right, right to left, or in some more arcane fashion. Let us stick with left-to-right order, unless specifically changed.

`map` on trees is given by the induced morphism into the pullback

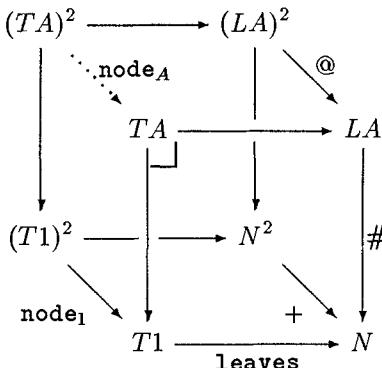
$$\begin{array}{ccccc} TA & \longrightarrow & LA & & \\ \downarrow & \lrcorner & \downarrow & \searrow & \\ TB & \xrightarrow{Tf} & LB & \xrightarrow{Lf} & \\ \downarrow & \lrcorner & \downarrow & \downarrow & \downarrow \# \\ T1 & \xrightarrow{\text{id}} & N & \xrightarrow{\text{id}} & N \\ & & \searrow & \swarrow & \\ & & T1 & \xrightarrow{\text{leaves}} & N \end{array}$$

Note that the shape has remained fixed while the data changes. Of course, this construction generalises to define `map` for any shapely type constructor.

Other operations change the shape and leave the data fixed. For example, the balancing of a binary tree is given by an operation $\text{bal}_A : TA \rightarrow TA$ which is independent of the labels



Finally, both data and shape may change together, as occurs with the node operation, which creates a tree from two subtrees.



The number of leaves in the result is the sum of those in the sub-trees, while the lists of leaves must be appended. Note that if **leaves** represents the leaves from right to left then the order of the arguments must be swapped to define $@$. In some sense, the choice of $@$ here fixes the representation of the leaves.

Trees whose nodes are also labelled are shapely, too. The defining pullback is given by

$$\begin{array}{ccc}
 T(A, B) & \xrightarrow{\quad} & LA \times LB \\
 \downarrow & & \downarrow \# \times \# \\
 T1 & \xrightarrow{\langle \text{leaves}, \text{nodes} \rangle} & N \times N
 \end{array}$$

Most of the usual first-order data types of functional languages can be constructed in this way, and arrays are available within the same framework, too. Note that, since shapely type constructors are always covariant functors, contravariant constructions, such as function types, cannot be shapely.

4.1 An Alternative Approach

In defining shapely type constructors, the data is given by a tuple of lists, one for each kind of data whose type is $LA_1 \times LA_2 \times \dots \times LA_n$. What happens if these lists are interleaved to resemble input strings, of type $L(A_1 + A_2 + \dots + A_n)$? In other words, one could consider the functors which are shapely over $L\Sigma$ instead of ΠL . In fact, the resulting class of functors is unchanged, as shown by the following

Proposition 6. ΠL^m and $L\Sigma$ are each shapely over the other.

Proof. Clearly, there is shapely natural transformation $\Pi L^m \Rightarrow L\Sigma$ obtained by concatenation. Conversely, define the natural transformation $\text{check}_{A,B}$ by

$$(A + B) \times LA \xrightarrow{d} (A \times LA) + (B \times LA) \xrightarrow{[\text{cons}, \pi']} LA$$

It is shapely in A whence $\kappa_1 = \text{foldr}(\text{nil}, \text{check})$ is, too. This can be generalised to define $\kappa_i : L(A_1 + A_2 + \dots + A_n) \Rightarrow LA_i$ which strips from a list all entries which are not from A_i . It is shapely in A_i . The obvious n -fold generalisation of Proposition 2 shows that

$$\kappa = \langle \kappa_i \rangle : L\Sigma \Rightarrow \Pi L^m$$

is a shapely transformation. \square

5 Shape Polymorphism

Shape polymorphism arises when the operations on the shape and on the data are independent of each other. Currently, such operations must redefined for each new type, increasing the bulk of the code and reducing clarity. Aside from `map`, the shape polymorphic operations include

$$\text{zip} : FA \times_{\#} FB \rightarrow F(A \times B) \quad (6)$$

$$\tau : FA \times B \rightarrow F(A \times B) \quad (7)$$

$$\tau' : A \times FB \rightarrow F(A \times B) \quad (8)$$

$$\tau'' = F\tau' \circ \tau : FA \times FB \rightarrow F^2(A \times B) \quad (9)$$

$$\text{copies} : F1 \times A \rightarrow FA \quad (10)$$

$$\text{square} = \tau \circ \langle \#, \text{id} \rangle : FA \rightarrow F^2 A . \quad (11)$$

τ' is dual to τ and the operation `copies` instantiates all entries of the shape to the given value, while `square` replaces each entry with a copy of the whole.

Closely related to `map` are the *pointwise operators* introduced by example in [Jon90] and defined in [Jay93a]. These iterate an endomorphism at each entry in a shape. The number of iterations at each entry is determined by a *weight* on the shape i.e. a morphism $F1 \rightarrow FN$. Particular shapes may have special weights (e.g. one can weight each leaf in a tree by its depth) but weights on lists yield shape polymorphic operations. Examples include weighting each entry by the length of the list, or by its position. Their use in defining the discrete Fourier transform *op. cit.* shows it to be shape polymorphic.

6 Shapely Operations

In general computation, the shape of the result is influenced by that of the data. Examples include filtering of a list, or graph reduction. There is, however, a large body of computations where the shape of the result depends only on that of the input, without reference to the data. As well as the shape polymorphic operations, these include many operations where the shape affects the data, but not conversely. Examples include averaging of entries, pointwise operators, and many algorithms, such as the DFT.

For such *shapely operations* it is profitable to separate the internal representations of shape and data. Then shape processing can be treated as part of compilation, in which shape errors are detected (e.g. attempting to `zip` two different shapes), the shape of the result is computed, and any shape information required by the data is supplied. Data can then be stored and processed in arrays. In this way, the clarity of type (and shape!)-checking is combined with the efficiency of array-processing.

The main points are illustrated by the decomposition of a tree into either a leaf or a pair of sub-trees.

$$\begin{array}{ccccc}
 TA & \xrightarrow{\sim} & A + (TA)^2 & \xrightarrow{\text{id} + \delta^2} & A + (LA)^2 & \xrightarrow{[\eta, @]} & LA \\
 \downarrow & & \downarrow & & \downarrow & & \downarrow \\
 T1 & \xrightarrow{\sim} & 1 + (T1)^2 & \xrightarrow{\text{id} + \delta^2} & 1 + N^2 & \xrightarrow{[\text{one}, +]} & N
 \end{array}$$

The shape of the result is determined by that of the input, but in order to know where to break the list of leaves, the number n of leaves in the left subtree is required. Shape processing would add the computed value of n to the environment prior to the array-processing.

7 Initial Algebras

7.1 Endofunctors

Let F be shapely over L . The initial algebra F_0 will be constructed as an object of well-formed expressions. The alphabet is given by $\Omega = F1$. The well-formed expressions are described by a pullback

$$\begin{array}{ccc}
 F_0 & \xrightarrow{\phi} & L\Omega \\
 \downarrow & & \downarrow \chi_1 \\
 1 & \xrightarrow{\langle \text{nil}, \text{one} \rangle} & L\Omega \times N
 \end{array}$$

where χ_1 attempts to recognise the well-formed expressions. More precisely, its second component counts the number of well-formed expressions created, while the first component yields that part of the input string (if any) which could not be recognised.

Observe that $! : F1 \rightarrow 1$ makes 1 an F -algebra. The recogniser χ_1 is a special case of a more general “parser”

$$\chi_C : L\Omega \rightarrow L\Omega \times LC$$

to be defined for any F -algebra $\gamma : FC \rightarrow C$. The terminology is motivated by the case when C represents the parse trees, as given by F_0 below.

$\chi_C = \text{foldr}((\text{nil}, \text{nil}), \theta_C)$ where $\theta_C : \Omega \times L\Omega \times LC \rightarrow L\Omega \times LC$ performs one step of the parse, as will now be described.

First, the middle component of the source is nil unless the parse has already failed. That is, we can re-express the source (using the appropriate isomorphism) as $(\Omega \times LC) + (\Omega \times \Omega \times L\Omega \times LC)$ and then $\theta_C = [\zeta_C, \text{cons} \circ (\text{id} \times \text{cons}) \times \text{id}]$ where ζ_C remains to be described, by cases.

Lemma 7. *The test $\text{Eq} \circ (\pi, \text{take}) \circ (\delta \times \text{id}) : \Omega \times LC \rightarrow \text{bool}$ recognises the sub-object*

$$(\text{id} \times @) \circ ((\#, \delta) \times \text{id}) : FC \times LC \rightarrow \Omega \times LC .$$

Proof. In brief, the test picks out those pairs where the arity of the Ω is no greater than the length of the list. In that case, we have the resources to construct something of type FC with a list of C 's left over. \square

Let $\iota' : QC \rightarrow \Omega \times LC$ be the pullback of the above test along false . Then ζ_C is given by decomposing $\Omega \times LC$ into the specified coproduct followed by

$$[(\text{nil}, \text{cons} \circ (\gamma \times \text{id})), (\eta \times \text{id}) \circ \iota'] : (FC \times LC) + QC \rightarrow L\Omega \times LC .$$

Lemma 8. *If $h : (C, \gamma) \rightarrow (C', \gamma')$ is an F -algebra homomorphism then*

$$(\text{id} \times Lh) \circ \chi_C = \chi_{C'} .$$

Proof. Clearly Q is a functor and ι' is a natural transformation. Hence ζ, θ and χ are natural with respect to F -algebra homomorphisms. \square

Hence, the construction of F_0 can be given in stages by

$$\begin{array}{ccc} F_0 & \xrightarrow{\phi} & L\Omega \\ \downarrow & \lrcorner & \downarrow \\ h_C & \downarrow & \downarrow \chi_C \\ C & \xrightarrow{(\text{nil}, \eta_C)} & L\Omega \times LC \\ \downarrow & \lrcorner & \downarrow \\ 1 & \xrightarrow{(\text{nil}, \text{one})} & L\Omega \times N \end{array}$$

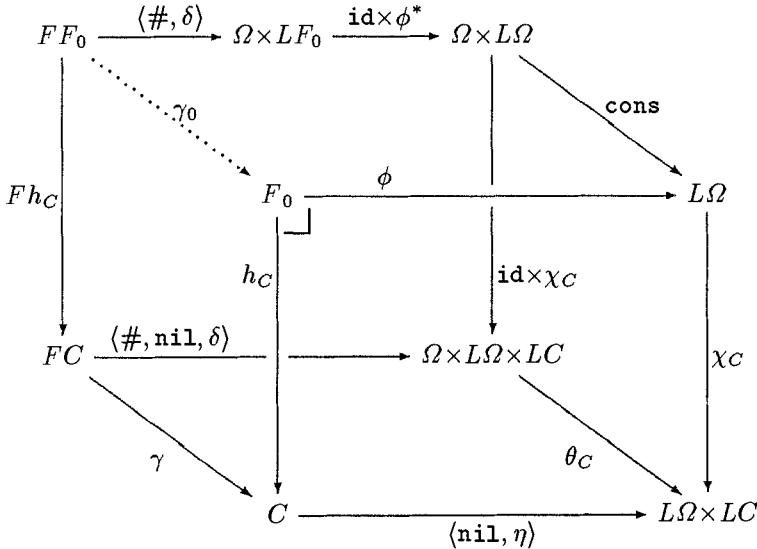
$$\text{id} \times \#$$

Now we will show that F_0 is an initial F -algebra with h_C the unique algebra homomorphism to C .

Lemma 9. $\chi_C \circ \phi^* = (\text{nil}, Lh_C) : LF_0 \rightarrow L\Omega \times LC$

Proof. It suffices to show that both sides of the equation are foldright of (nil, nil) and $\text{foldr}(\text{id}, \theta_C) \circ (\phi \times \text{id})$. The `cons` case for the right-hand-side is not trivial. See [JC94] for details. \square

Consider the following cube.



Lemma 9 implies the commutativity of its rear face. The right and bottom faces commute by the definitions of χ and θ . Hence, there is an induced F -action γ_0 that makes h_C a homomorphism. (Of course, the definition of γ_0 and its action is not dependent on the particular choice of C , since we can always work over the algebra $C = 1$.)

It remains to prove its uniqueness. Let $h : F_0 \rightarrow C$ be any F -algebra homomorphism. Then the pullback defining h_C factorises it as $h \circ h_0$ as in Fig. 1. (We abuse notation by denoting h_{F_0} by h_0 and χ_{F_0} by χ_0 etc.) Hence it suffices to prove that $h_0 = \text{id}$.

The following lemma shows that parsing into F_0 is reversible.

Lemma 10. $@ \circ (\text{id} \times \phi^*) \circ \chi_0 = \text{id}_{L\Omega}$ \square

Hence

$$\phi \circ h_0 = @ \circ (\text{id} \times \phi^*) \circ (\text{nil}, \eta) \circ h_0 \quad (12)$$

$$= @ \circ (\text{id} \times \phi^*) \circ \chi_0 \circ \phi \quad (13)$$

$$= \phi \quad (14)$$

and so $h_0 = \text{id}$ since ϕ is a monomorphism.

$$\begin{array}{ccc}
F_0 & \xrightarrow{\phi} & L\Omega \\
h_0 \downarrow & \lrcorner & \downarrow \chi_0 \\
F_0 & \xrightarrow{\langle \text{nil}, \eta \rangle} & L\Omega \times L F_0 \\
h \downarrow & \lrcorner & \downarrow \text{id} \times Lh \\
C & \xrightarrow{\langle \text{nil}, \eta \rangle} & L\Omega \times LC
\end{array}$$

Fig. 1. Factorisation of h_C

7.2 The General Case

A functor $F : \mathcal{C}^m \times \mathcal{C}^n \rightarrow \mathcal{C}^n$ can be used to represent a system of (parametrised) domain equations [SP82], whose solution is can be found by constructing, for each object A in \mathcal{C}^m , an initial algebra $\alpha_A : F(A, F^\dagger A) \rightarrow F^\dagger A$ for the functor $F(A, -)$. If such always exist, then F^\dagger extends to a functor whose action on $f : A \rightarrow B$ is the $F(A, -)$ -algebra homomorphism induced by the action

$$F(A, F^\dagger B) \xrightarrow{F(f, \text{id})} F(B, F^\dagger B) \xrightarrow{\alpha_B} F^\dagger B$$

Further, if $\beta : F(\text{id}, G) \Rightarrow G : \mathcal{C}^m \rightarrow \mathcal{C}^n$ is a natural transformation, then the unique algebra homomorphisms induce a natural transformation $\beta^\dagger : F^\dagger \Rightarrow G$.

Theorem 11. *If $\phi : F \Rightarrow \Delta \Pi L^{m+n} : \mathcal{C}^m \times \mathcal{C}^n \rightarrow \mathcal{C}^n$ is a shapely type constructor then F^\dagger exists and is one, too. Further, if $\beta : F(\text{id}, G) \Rightarrow G : \mathcal{C}^m \rightarrow \mathcal{C}^n$ is a shapely transformation, then so is β^\dagger .*

Proof. F is determined by its projections onto \mathcal{C} which are all shapely over ΠL^{m+n} . By the Bekic Lemma, we can treat these individually, or, equivalently, assume that $n = 1$. Then for each object A in \mathcal{C}^m the initial algebra $F^\dagger A$ for $F(A, -)$ is constructed as above.

The cartesian-ness of β^\dagger follows from that of χ_G which, by Theorem 3, reduces to that of θ_G . Examination of the cases reduces this to the cartesian-ness of cons , η and β .

The strength for F^\dagger is defined using the defining pullback for $F^\dagger(A \times B)$ and the strength of $LF(A, 1) \times LGA$. It follows that β^\dagger is shapely.

Now, taking β to be

$$F(A, \Pi L^m A) \xrightarrow{\phi} \Pi L^m A \times \Pi L^m A \cong \Pi(LA \times LA) \xrightarrow{\Pi @} \Pi L^m A$$

(where \cong permutes the arguments) induces a shapely transformation $F^\dagger \Rightarrow \Pi L^m$ which shows that F^\dagger is a shapely type constructor. \square

8 Related and Further Work

Banger and Skillicorn [BS93] give a categorical semantics for arrays, which are represented by their dimensions and some unbounded lists. They do not use pullbacks to represent their types, nor is there a general theory of shape.

Those shapely types, such as matrices, which are not part of the usual functional programming approach, can be represented using dependent types, but their type checking must be performed dynamically, whereas most shape computation should be performed statically. The exact relationship to dependent types, and the internal logic of locoses, require further exploration.

Further work will consider how to incorporate shape ideas into programming language design, and compiler construction, using classes in a functional or object-oriented setting.

References

- [BS93] C.R. Banger and D.B. Skillicorn. A foundation for theories of arrays. Queen's University, Canada, 1993.
- [BW88] R. Bird and P. Wadler. *Introduction to Functional Programming*. International Series in Computer Science. Prentice Hall, 1988.
- [CF92] J.R.B. Cockett and T. Fukushima. About **charity**. University of Calgary preprint, 1992.
- [CLW93] A. Carboni, S. Lack, and R.F.C. Walters. Introduction to extensive and distributive categories. *Journal of Pure and Applied Algebra*, 84:145–158, 1993.
- [Coc90] J.R.B. Cockett. List-arithmetic distributive categories: locoi. *Journal of Pure and Applied Algebra*, 66:1–29, 1990.
- [CS92] J. R. B. Cockett and D. Spencer. Strong categorical datatypes. In R. A. G. Seely, editor, *International Meeting on Category Theory 1991*, Canadian Mathematical Society Proceedings. American Mathematics Society, Montreal, 1992.
- [Jay93a] C.B. Jay. Matrices, monads and the fast fourier transform. Technical Report UTS-SOCS-93.13, University of Technology, Sydney, 1993.
- [Jay93b] C.B. Jay. Tail recursion through universal invariants. *Theoretical Computer Science*, 115:151–189, 1993.
- [JC94] C.B. Jay and J.R.B. Cockett. Shapely types and shape polymorphism: Extended version. Technical Report UTS-SOCS-94-??, University of Technology, Sydney, 1994.
- [Jon90] G. Jones. Deriving the fast fourier transform algorithm by calculation. In *Functional programming, Glasgow 1989*, Springer Workshops in Computing. Springer Verlag, 1990.
- [Man92] E. Manes. *Predicate Transformer Semantics*, volume 33 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.
- [SP82] M. Smith and G. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal of Computing*, 11, 1982.

Bottom-up Grammar Analysis — A Functional Formulation —

Johan Jeuring*and Doaitse Swierstra

Utrecht University

P.O.Box 80.089, 3508 TB Utrecht, The Netherlands

email: {johan,doaitse}@cs.ruu.nl

Abstract

This paper discusses bottom-up grammar analysis problems such as the `EMPTY` problem and the `FIRST` problem. It defines a general class of bottom-up grammar analysis problems, and from this definition it derives a functional program for performing bottom-up grammar analysis. The derivation is purely calculational, using theorems from lattice theory, the Bird-Meertens calculus, and laws for list-comprehensions. Sufficient conditions guaranteeing the existence of a solution emerge as a byproduct of the calculation. The resulting program is used to construct programs for the `EMPTY` problem and the `FIRST` problem.

1 Introduction

Grammar analysis is performed in many different situations: Yacc tests whether or not its input grammar is LALR(1), parser generators contain functions for determining whether or not a nonterminal can derive the empty string (`EMPTY`) as part of determining the set of all symbols that can appear as the first symbol of a derived string (`FIRST`), and for determining the set of symbols that can appear as the first symbol following upon a string derived by a given nonterminal (`FOLLOW`). Other, similar, problems arise when analysing attribute dependencies in attribute grammars: determine the inherited attributes upon which a synthesised attribute depends (`IS`), and, conversely, determine the synthesised attributes upon which an inherited attribute depends (`SI`). Such problems are called *grammar analysis problems*.

Grammar analysis problems can be divided into two classes: *bottom-up* and *top-down*. The difference between these classes is that the required information for a nonterminal in a top-down problem depends on the possible contexts for that nonterminal, whereas in a bottom-up problem the contexts of a nonterminal can be ignored. Often the output of a bottom-up problem is used in a top-down problem.

*address from January 1, 1994: Chalmers Tekniska Högskola, Institutionen för Datavetenskap,
S-412 96 Göteborg, Sweden

The specification of a grammar analysis problem determines the class to which it belongs: `EMPTY`, `FIRST`, and `IS` are bottom-up grammar analysis problems, the `FOLLOW` and `SI` problems belong to the top-down class. This paper studies bottom-up grammar analysis.

Grammar analysis problems are described by sets of mutually recursive equations, and the solution of a grammar analysis problem is a fixed point of this equational system. Möncke and Wilhelm [11] observe this, and give several solutions, depending on the conditions that are satisfied, for such problems. The goal of this paper is to derive the solutions given by Möncke and Wilhelm. We start with a very general specification of a bottom-up grammar analysis problem, and we derive a function of which the fixed point gives the solution of the problem. This function is obtained by applying laws to components of the expressions occurring in the specification. The laws we apply are familiar laws for, for example, list-comprehensions [13], and maps [1, 9]. Sufficient conditions for guaranteeing the existence of a fixed point solution emerge as a byproduct of this derivation. An important advantage of a derivation of a program is that it is clear why and where conditions are imposed upon the components of the program. Finally we give the implementation of the derived algorithm in the functional language Gofer [7, 3]. Incorporating the functions for solving grammar analysis problems in parser generators such as a functional version of Yacc [12], Ratatosk [10], and Happy [4] would reduce the amount of code used in these parser generators.

This paper is organised as follows. Section 2 defines the datatypes that are used in manipulating grammars in Gofer. Section 3 introduces the concepts of lattice theory needed in the subsequent sections. Section 4 defines the class of bottom-up grammar analysis problems, and gives some examples. Section 5 sketches a derivation of an algorithm that can be used to solve bottom-up problems. The complete derivation is given in [6]. Section 6 gives an implementation in Gofer of the datatypes and functions constructed in the previous sections. Section 7 concludes the paper.

2 Datatypes for Grammars in Gofer

This section defines various datatypes in Gofer used in analysing and representing grammars. Most of the non-standard notation is taken from the Bird-Meertens calculus [1].

Functions

Projection exl (expr) selects the left (right) component of a pair. Given functions $f : A \rightarrow B$ and $g : A \rightarrow C$, function $f \Delta g : A \rightarrow B \times C$ (split) applies both f and g to an argument: $(f \Delta g) a = (f a, g a)$. The type $B \times C$ is the cartesian product of the sets B and C . Given functions $f : A \rightarrow B$ and $g : C \rightarrow D$, function $f \times g : A \times C \rightarrow B \times D$ (product) applies f to the first component, and g to the second component of its argument: $(f \times g) (a, c) = (f a, g c)$. The laws for compositions of projections, split, and product are omitted. Function application binds stronger than a binary operator; composition binds weakest.

Lists

The datatype (finite) `list` is a prominent datatype in the subsequent sections, and

we will use a number of properties that are satisfied by functions defined on the datatype *list*. The empty list is denoted by $[]$, and the concatenation of two lists x and y is denoted by $x \dagger y$. Prepending an element x to a list xs is denoted by $x : xs$. The datatype *list* over base type A is denoted by A^* . For $f : A \rightarrow B$, function $f^* : A^* \rightarrow B^*$, called a *map* function takes a list and applies function f to all elements in the list, so $f^* xs = [f x \mid x \leftarrow xs]$. For the map function we have

$$\begin{aligned} f^* [] &= [] \\ f^* (x \dagger y) &= f^* x \dagger f^* y \\ f^* (x : xs) &= f x : f^* xs \end{aligned} \tag{1}$$

Map-distributivity says that the composition of two maps is a map again, i.e., for all functions f and g :

$$f^* \cdot g^* = (f \cdot g)^* \tag{2}$$

Furthermore, the result of mapping the identity function over an argument is the argument itself, so $id_{A^*} = id_{A^*}$. These equalities say that $*$ is a *functor*. An important functional programming construct we use is *list-comprehension*. For example,

$$[(x, y) \mid x \leftarrow [1, 2], y \leftarrow [3, 4]] = [(1, 3), (1, 4), (2, 3), (2, 4)]$$

We will use the following laws for list-comprehensions [13] in some calculations.

$$[t \mid t \leftarrow ts] = ts \tag{3}$$

$$[f t \mid q] = f^* [t \mid q] \tag{4}$$

$$[t \mid p, q] = concat [[t \mid q] \mid p] \tag{5}$$

where function $concat : A^{**} \rightarrow A^*$ flattens a list of lists. Function $concat$ is defined in terms of the reduce operator. The *reduce* operator $/$ takes an associative operator \oplus with unit 1_{\oplus} , and a list, and places the operator in between the elements of a list, so $\oplus / [a, b, c] = a \oplus b \oplus c$. For operator $\oplus : A \times A \rightarrow A$ we have $\oplus / : A^* \rightarrow A$. It can be defined by

$$\begin{aligned} \oplus / [] &= 1_{\oplus} \\ \oplus / (x \dagger y) &= (\oplus / x) \oplus (\oplus / y) \\ \oplus / (x : xs) &= x \oplus (\oplus / xs) \end{aligned} \tag{6}$$

Function $concat$ is an example of a reduce: it is defined by $concat = \dagger /$. For the composition of a map and function $\dagger /$, and for the composition of a reduce, and function $\dagger /$ we have

$$f^* \cdot \dagger / = \dagger / \cdot f^{**} \tag{7}$$

$$\oplus / \cdot \dagger / = \oplus / \cdot (\oplus /)^* \tag{8}$$

Terminals and Nonterminals

Suppose a terminal is a value of type b , and a nonterminal is a value of type a . A

symbol that is either a nonterminal of type a or a terminal of type b is a value of the datatype *Symbol* defined as

$$\text{data Symbol } a\ b = N\ a \mid T\ b$$

An element $N\ x$ is considered to be a nonterminal, and an element $T\ y$ is considered to be a terminal.

Grammars

A context-free grammar consists of sets of nonterminals, terminals, productions, and a start-symbol. In Gofer, the sets of nonterminals and terminals correspond with the types a and b , respectively. These types are parameters of the definition of a context-free grammar. We represent a context-free grammar in Gofer by a pair, the first component of which denotes the start-symbol, and the second component of which denotes the productions of the grammar. The start-symbol is a nonterminal, i.e., a value of type a . The productions of a grammar are a set of pairs the left-component of which is a non-terminal, and the right component of which is a list of symbols. A context-free grammar is a value of the type *Grammar*, which is defined by

$$\text{type Grammar } a\ b = (a, [(a, [Symbol\ a\ b])])$$

Function *rhss* takes a grammar and a nonterminal nt and returns the right-hand sides of the productions of nt . It is defined by

$$\text{rhss } g\ nt = [\text{rhs} \mid (nt, \text{rhs}) \leftarrow \text{expr } g]$$

Function *nts* takes a grammar, and returns the list of nonterminals of the grammar. We assume that for each nonterminal there exists at least one production. Let function *rmdups* remove duplicates from a list, then function *nts* is defined by

$$\text{nts } g = \text{rmdups } (\text{exl* } (\text{expr } g))$$

Parse Trees

To determine whether or not the empty string can be derived from a nonterminal (the *EMPTY* problem), we have to refer to all sentences that are derivable from the given nonterminal in the given grammar. A derivation using productions of a context-free grammar corresponds to a *parse tree* or *derivation tree*, i.e., an element of the datatype *Rosetree*, which is defined by

$$\text{data Rosetree } a\ b = \text{Node } a\ [\text{Rosetree } a\ b] \mid \text{Leaf } b$$

Suppose function *top* : *Rosetree* $a\ b \rightarrow \text{Symbol } a\ b$ returns the top of a rose-tree. For each subtree of a derivation tree of the form *Node* $a\ x$ we have that $a \rightarrow \text{top* } x$ is a production of the grammar.

The function *sen* takes a rose-tree, and returns the sentence of which the rose-tree is a derivation. Function *sen* is defined by

$$\text{sen } (\text{Node } a\ x) = +/ (\text{sen* } x)$$

$$\text{sen } (\text{Leaf } b) = [b]$$

Catamorphisms on Rose-Trees

For many kinds of recursive datatypes we can define a class of functions, called catamorphisms, which recursively replace constructors by functions [8]. By definition, a *catamorphism* on the datatype *Rosetree* is a function $h : \text{Rosetree} \rightarrow c$ that is uniquely determined by functions f and g as follows.

$$\begin{aligned} h(\text{Node } a \ x) &= f a (h* x) \\ h(\text{Leaf } b) &= g b \end{aligned}$$

For such a function h we write $h = \langle f, g \rangle$. The function *sen* defined above is a catamorphism, i.e.,

$$\text{sen} = \langle \lambda x \rightarrow \text{++}/x, \lambda b \rightarrow [b] \rangle$$

Another example of a catamorphism on *Rosetree* is the *height* function, which returns the height of a rose-tree. The definition of function *height* is omitted. We will encounter several other catamorphisms on rose-trees in the following sections.

3 Lattice Theory

This section only gives the definitions of notions from lattice theory that are used in the subsequent sections. For a more extensive introduction to lattice theory the reader is referred to e.g. [2].

Lattices and CPO's

A *partial* order on a set A is a reflexive, antisymmetrical, and transitive binary relation on A . A *partially ordered set* or *poset* is a pair (D, \sqsubseteq) consisting of a set D together with a partial order \sqsubseteq on D . If it exists, the least or *bottom* element of a poset is usually denoted by \perp . Given $d, d' \in D$, their *join*, denoted by $d \sqcup d'$, is the least element in D that is greater than both d and d' . It is fully characterised by the following equation:

$$c = d \sqcup d' \equiv (\forall e :: c \sqsubseteq e \equiv d \sqsubseteq e \wedge d' \sqsubseteq e)$$

Note that the join of two elements in D is uniquely defined when it exists. The *least upperbound* or *lub* of a subset $X \subseteq D$ is denoted by \sqcup/X . It is defined by

$$c = \sqcup/X \equiv (\forall e :: c \sqsubseteq e \equiv (\forall x : x \in X : x \sqsubseteq e))$$

Not every $X \subseteq D$ needs to have a lub. Let (D, \sqsubseteq) be a poset. If for all elements d and d' their join $d \sqcup d'$ exists, then (D, \sqsubseteq) is called a *join semilattice*. Let S be a subset of a poset. S is said to be *directed* if every finite subset of S has an upper bound. A poset D is a *complete partial order* or *CPO* if it contains a bottom element, and if each directed subset of D has a lub, so \sqcup/X exists for all directed subsets $X \subseteq D$.

Fixed Points

An element $d \in D$ is a *fixed point* of function $f : D \rightarrow D$ if $f d = d$. It is a *least fixed point* if for any other fixed point d' of f we have $d \sqsubseteq d'$. A function $f : D \rightarrow E$

is *monotonic* if it respects the ordering on D , i.e., $d \sqsubseteq d' \Rightarrow f d \sqsubseteq f d'$. A function $f : D \rightarrow E$ is *continuous* if it respects lubs of directed subsets, i.e., if $X \subseteq D$ is a directed subset, then $(f \cdot \sqcup /) X = (\sqcup / \cdot f*) X$.

Let D be a finite set, (D, \sqsubseteq) a CPO with bottom \perp , and $g : D \rightarrow D$ a continuous function. It follows from the CPO Fixed Point Theorem I [2] that function g has a least fixed point μg , defined by $\mu g = \sqcup / [g^n \perp \mid n \leftarrow [0..]]$. Since $g^i \perp \sqsubseteq g^{i+1} \perp$, we have that the least fixed point of g equals the first element in $[g^n \perp \mid n \leftarrow [0..]]$ that occurs twice, i.e., $\sqcup / [g^n \perp \mid n \leftarrow [0..]] = \text{lfp } g \perp$ where function lfp is defined by

$$\text{lfp } f x = \begin{cases} x & \text{if } f x = x \\ \text{lfp } f (f x) & \text{otherwise} \end{cases}$$

The *Fixed Point Fusion Theorem* (or Plotkin's Lemma) is used to reason about fixed points. This theorem reads as follows.

$$f \perp = \perp \wedge f \cdot h = g \cdot f \Rightarrow f \mu h = \mu g$$

We use the Fixed Point Fusion Theorem and the CPO Fixed Point Theorem I as follows. Consider the function $(+1)$. Define $\infty = \mu(+1)$. Taking $h = (+1)$ and writing 0 for the bottom \perp of natural numbers, we get, applying the Fixed Point Fusion Theorem,

$$f 0 = \perp \wedge f (n+1) = g (f n) \Rightarrow f \infty = \mu g$$

4 Grammar Analysis Problems

Although in some grammar analysis problems only a property of the start-symbol of the grammar is sought, we define a grammar analysis problem to be a problem which requires finding information about all nonterminals of the grammar. This section defines bottom-up grammar analysis problems. The first subsection gives some examples of grammar analysis problems. The second subsection discusses functions for generating derivation trees. The third subsection defines bottom-up grammar analysis problems.

4.1 Examples of Grammar Analysis Problems

Part of determining whether or not a grammar is LL(1) consists of solving the grammar analysis problems **EMPTY**, **FIRST**, and **FOLLOW**.

EMPTY

Given a grammar g and a nonterminal nt from g , the expression $\text{Empty } g nt$ is a boolean expressing whether or not it is possible to derive the empty string from nt , using the productions from g . $\text{Empty } g nt$ is defined by

$$\text{Empty } g nt = [] \neq [x \mid nt \xrightarrow{*} x, x = []]$$

where $\xrightarrow{*}$ denotes a derivation with productions from g .

FIRST

Given a grammar g and a nonterminal nt from g , the expression $\text{First } g nt$ is the set

of terminals (the set of terminals is the set X) that can appear as the first element of a string of terminals derivable from nt . It is defined by

$$\text{First } g \text{ } nt = \text{ rmdups } [a \mid nt \xrightarrow{*} [a] \# x, x \in X^*]$$

FOLLOW

Given a grammar g and a nonterminal nt from g , the expression $\text{Follow } g \text{ } nt$ is the set of terminals that can follow on nt in a derivation starting with the start-symbol S from g . It is defined by

$$\text{Follow } g \text{ } nt = \text{ rmdups } [a \in X \mid S \xrightarrow{*} u \# [nt, a] \# v]$$

Bottom-up versus Top-down

The definitions in the first two examples given above require finding information about a nonterminal, and do not refer to the context in which such a nonterminal appears. These two examples are bottom-up grammar analysis problems. The definition in the last example explicitly refers to the context in which the nonterminal appears, namely $u \# [., a] \# v$. This example is a top-down grammar analysis problem. In the rest of the paper we limit ourselves to bottom-up problems.

4.2 Generating Trees

The definitions in the examples of grammar analysis problems given in the previous subsection typically refer somehow to all sentences derivable from a nonterminal. The sentences derivable from a nonterminal can be obtained from the derivation trees of the grammar with the given nonterminal in the root. In this subsection we define a function returning all possible derivation trees of a grammar.

Function *generate* takes a grammar, and returns a list of lists, in which each list contains all derivation trees with the same nonterminal in the root. Before we give the definition, we discuss the function *cp* (cartesian product), which is used in the definition of function *generate*.

Function *cp*

Function *cp* returns the cartesian product of a list of lists. It is defined as a map followed by a reduce by

$$\begin{aligned} cp &= \lambda / \cdot [\cdot]^{**} \\ xs \lambda ys &= [x \# y \mid x \leftarrow xs, y \leftarrow ys] \end{aligned}$$

where $[\cdot]$ takes an element a , and returns the singleton list containing that element: $[a]$. Note that $[[\cdot]]$ is the unit of operator λ . Function *cp* commutes with function f^{**} for all functions f , i.e., for all functions f we have

$$f^{**} \cdot cp = cp \cdot f^{**} \tag{9}$$

Function *generate*

Function gh is defined in the context of a grammar g , which from now on is considered a constant. It takes a natural number n , and a symbol s , and returns the collection of all derivation trees, of height at most n , derivable with the productions of g with symbol s in the root, so

$$gh\ n\ s = [y \mid s \xrightarrow{*} y \wedge \text{height } y \leq n]$$

where we suppose that $\xrightarrow{*}$ derives derivation trees instead of strings with productions from grammar g . Function *generate* is defined in terms of function gh as follows.

$$\text{generate } g = (gh\ \infty \cdot N)^* (nts\ g)$$

Function gh can be defined recursively in various ways; we have chosen the following definition which is easily manipulated in calculations. Function gh is defined by pattern matching on its arguments. There are no trees of height zero, so $gh\ 0\ symbol = []$. There is just one derivation tree of height at most $n+1$ that can be built from a terminal: $gh\ (n+1)\ (T\ b) = [\text{Leaf } b]$. The list of derivation trees of height at most $n+1$ derivable from a nonterminal nt contains the list of the derivation trees of height at most n derivable from nt . Furthermore, for each production for nt we add the cartesian product of the derivation trees of height at most n of the symbols of the right-hand side of a production for nt ; each element of the cartesian product is turned into a derivation tree using function *Node* nt .

$$\begin{aligned} gh\ (n+1)\ (N\ a) &= (gh\ n\ (N\ a)) \uplus \\ &\quad [\text{Node } a\ c \mid rhs \leftarrow rhss\ g\ a, c \leftarrow cp\ ((gh\ n)^*\ rhs)] \end{aligned} \tag{10}$$

We do not bother about duplicate elements in $gh\ n\ s$; applying function *rmdups* to the right-hand expression of the last equation would have removed them. The right-hand side argument of \uplus in the last equation of the definition of function gh can be rewritten using laws for list-comprehensions. The resulting equality will be used in the calculation in Section 5.

$$\begin{aligned} &[\text{Node } a\ c \mid rhs \leftarrow rhss\ g\ a, c \leftarrow cp\ ((gh\ n)^*\ rhs)] \\ &= \text{equations (5), (3), and (4) for list-comprehensions} \\ &\quad ((\text{Node } a)^* \cdot \uplus / \cdot cp^* \cdot (gh\ n)^*)\ (rhss\ g\ a) \end{aligned}$$

4.3 Bottom-up Problems

We formalise the notion of a grammar analysis problem. In case of the *EMPTY* problem, we want to determine for all nonterminals nt from a grammar g whether or not it is possible to derive the empty string from nonterminal nt . A non-executable specification for this problem reads as follows. Given a nonterminal nt we apply a function p to each derivation tree with nt in the root. Function p determines whether or not the string represented by the derivation tree is empty, i.e., $p = ([] =) \cdot \text{sen}$. Note that function p corresponds with the two expressions $nt \xrightarrow{*} x, x = []$ occurring in the list-comprehension in the definition of *Empty* $g\ nt$. To determine whether or not it is possible to derive the empty string from nonterminal nt , we apply the function *combine* to the list of results obtained by applying function p to

all derivation trees with nt in the root. Function $combine$ is defined by $combine = ([] \neq)$, which equals the reduction $\vee/$. It corresponds with the expression in front of the list-comprehension in the definition of $Empty\ g\ nt$. Generalising this pattern, we now define the class of bottom-up grammar analysis problems.

(11) Definition A bottom-up grammar analysis problem, which analyses a grammar g with respect to a function $p : Rosetree\ a\ b \rightarrow c$, and an operator $\oplus : c \times c \rightarrow c$ with unit 1_{\oplus} , is an expression of the form $ag\ g\ p\ \oplus$, where function ag is defined as follows.

$$\begin{aligned} ag\ g\ p\ \oplus &= (id \Delta (af \cdot gh \infty \cdot N)) * (nts\ g) \\ \text{where } af &= combine \cdot properties \\ properties &= p* \\ combine &= \oplus/ \end{aligned}$$

In case of the bottom-up grammar analysis problems **EMPTY** and **FIRST** we write

$$\begin{aligned} empties\ g &= ag\ g\ (([] =) \cdot sen) \vee \\ firsts\ g &= ag\ g\ (take\ 1 \cdot sen) \cup \end{aligned}$$

where operator \cup is defined by $x \cup y = rmdups\ (x ++ y)$.

5 The Derivation of an Algorithm

Function ag can be implemented in a functional language, but executing $ag\ g\ p\ \oplus$ will result in a nonterminating computation because of the occurrence of ∞ in the definition of function ag . This section derives an algorithm that can be implemented as an always terminating program that returns the value of $ag\ g\ p\ \oplus$. To obtain this algorithm we use the lattice theory given in Section 3.

Replacing the constant ∞ by a variable n in the definition of function ag (Definition (11)) results in the following equality.

$$\begin{aligned} ag\ g\ p\ \oplus &= agn\ \infty \\ \text{where } agn\ n &= (id \Delta (af \cdot gh\ n \cdot N)) * (nts\ g) \end{aligned}$$

We use the CPO fixed point theorems to find the value of $agn\ \infty$ in finite time. Suppose there exists a function K such that for $n \geq 0$

$$agn\ (n+1) = K(agn\ n) \tag{12}$$

If we suppose furthermore that there exists a CPO (E, \sqsubseteq_E) with bottom $agn\ 0$, then the results in Section 3 show that if function $K : E \rightarrow E$ is continuous, then it has a least fixed point μK and $agn\ \infty = \mu K$.

The domains used in the grammar analysis problems are finite, that is, the target type E of function ag is a finite type. Since every finite join semilattice is a CPO, and since each monotonic function on a finite domain is continuous, it suffices to find a join semilattice with bottom $agn\ 0$, and a monotonic function K satisfying (12).

This section consists of three subsections. The first subsection constructs a join semilattice with bottom $agn\ 0$ for bottom-up grammar analysis problems. The second subsection derives a definition of a monotonic function K that satisfies equation (12). The third subsection discusses the conditions imposed upon the components of the bottom-up grammar analysis problem during the derivation of function K .

5.1 Constructing a Join Semilattice with Bottom $agn\ 0$

We want to construct a join semilattice (E, \sqsubseteq_E) with bottom $agn\ 0$ and join \sqcup_E . For that purpose, we impose our first condition on bottom-up grammar analysis problems.

We omit the calculation of the following equality for value $agn\ 0$.

$$agn\ 0 = (id \Delta 1_{\oplus}^{\bullet}) * (nts\ g)$$

where $a^{\bullet}\ b = a$ for all a and b . It follows that $agn\ 0$ is a list of length equal to the number of nonterminals of g , of which the second components are all equal to 1_{\oplus} . This suggests to construct the following join semilattice. Let E be the set of lists x of length equal to the number of nonterminals of g of which $exl* x = nts\ g$, and of which the second component of each element is an element of c , the result type of operator \oplus .

For the definition of the relation \sqsubseteq_E and the join \sqcup_E , we suppose that there exists a relation \sqsubseteq_c such that (c, \sqsubseteq_c) with join \sqcup_c is a join semilattice, and such that the unit 1_{\oplus} of operator \oplus occurring in the definition of a bottom-up grammar analysis problem is the bottom of c . Both the relation \sqsubseteq_E and the join \sqcup_E are now straightforward extensions of \sqsubseteq_c and \sqcup_c , respectively. Relation \sqsubseteq_E is defined by pairwise comparing elements with \sqsubseteq_c .

$$x \sqsubseteq_E y \equiv \text{and} (exr* x \ Upsilon_{\sqsubseteq_c} exr* y)$$

where function *and* is the reduce $\wedge /$, and where Υ_{\oplus} , with \oplus a binary function, zips two lists of equal length to a list of pairs, and then applies operator \oplus to all pairs in the list. The join of two elements is defined by pairwise joining the second components of the pairs.

$$x \sqcup_E y = exl* x \ Upsilon (exr* x \ Upsilon_{\sqcup_c} exr* y)$$

It is easy to prove that $agn\ 0$ is the bottom of E , using the fact that 1_{\oplus} is the bottom of c , and that (E, \sqsubseteq_E) is a join semilattice.

5.2 Finding Function K

In this subsection we derive a definition of a monotonic function K satisfying (12). It follows that $agn\ \infty$ is the least fixed point of function K , i.e., $agn\ \infty = \mu K$.

If we assume that there exists a J such that $af \cdot gh\ (n+1) \cdot N = J\ (agn\ n)$, then we easily calculate the following equality for $agn\ (n+1)$.

$$agn\ (n+1) = (id \Delta (J\ (agn\ n))) * (nts\ g)$$

It follows by abstracting from $agn\ n$ in the last expression that a function K satisfying (12) is defined by

$$K\ x = (id \Delta J\ x)*\ (nts\ g) \quad (13)$$

It remains to find a function J such that $af \cdot gh\ (n+1) \cdot N = J\ (agn\ n)$ holds.

Function J is obtained by manipulating the expression $af\ (gh\ (n+1)\ (N\ a))$, where a is an element of $nts\ g$. Applying the definitions of map and reduce, we have

$$\begin{aligned} af\ (gh\ (n+1)\ (N\ a)) &= af\ (gh\ n\ (N\ a)) \oplus \\ &\quad af\ [Node\ a\ c \mid rhs \leftarrow rhss\ g\ a, c \leftarrow cp\ ((gh\ n)*\ rhs)] \end{aligned}$$

We express the arguments of operator \oplus in the last expression above in terms of $agn\ n$ separately. By definition of $agn\ n$ we have

$$af \cdot gh\ n \cdot N = r\ (agn\ n) \quad (14)$$

if function r is defined by $r\ x\ a = at\ x\ a$, where function at returns the right component of the pair in x of which the left component equals a . This equation is used to express the left-hand argument of operator \oplus in terms of $agn\ n$. It remains to express the right-hand argument of operator \oplus in terms of $agn\ n$.

$$\begin{aligned} &af\ [Node\ a\ c \mid rhs \leftarrow rhss\ g\ a, c \leftarrow cp\ ((gh\ n)*\ rhs)] \\ &= \text{equality from Section 4} \\ &\quad (af \cdot (Node\ a)* \cdot +/ \cdot cp* \cdot (gh\ n)**) (rhss\ g\ a) \end{aligned}$$

If we can push af to the right within the map $(gh\ n)**$ in the composition of functions of the last expression in the above calculation, then we can use equation (14) again to obtain an expression of the desired form. Aiming at pushing af to the right then, we proceed with the composition of functions $af \cdot (Node\ a)* \cdot +/ \cdot cp* \cdot (gh\ n)**$. Abbreviate function $Node\ a$ to mt . Applying the definition of af , and equations (2), (7), and (8) we obtain

$$af \cdot mt* \cdot +/ \cdot cp* \cdot (gh\ n)** = \oplus / \cdot (\oplus / \cdot (p \cdot mt)* \cdot cp \cdot (gh\ n)*)*$$

At this point of the calculation we assume that there exists a function pn such that

$$p\ (Node\ nt\ x) = pn\ nt\ ((top \Delta p)*\ x) \quad (15)$$

$$p\ (Leaf\ x) = pl\ x \quad (16)$$

This condition is not unreasonable: for all *Rosetree* catamorphisms there exists such a function pn . We proceed the calculation with the expression within the map in the last expression of the above calculation.

$$\begin{aligned} &\oplus / \cdot (p \cdot mt)* \cdot cp \cdot (gh\ n)* \\ &= \text{assumption (15), equations (2), (9)} \\ &\quad \oplus / \cdot (pn\ nt)* \cdot cp \cdot ((top \Delta p)* \cdot (gh\ n))* \\ &= \text{assume equation (17) below} \\ &\quad H\ nt \cdot (id \Delta (af \cdot gh\ n))* \\ &= \text{introduction of function } r' \text{ below} \\ &\quad H\ nt \cdot (r'\ (agn\ n))* \end{aligned}$$

In this calculation we have assumed the existence of two functions: H , and r' such that a number of properties is satisfied. We have assumed the existence of a function H such that the following equality is satisfied.

$$\oplus / \cdot (pn \ nt) * \cdot cp \cdot ((top \ \Delta \ p) * \cdot (gh \ n)) * = H \ nt \cdot (id \ \Delta \ (af \cdot gh \ n)) * . \quad (17)$$

Finally, function r' is defined by

$$\begin{aligned} r' \ x \ (N \ a) &= (N \ a, r \ x \ a) \\ r' \ x \ (T \ b) &= (T \ b, pl \ b) \end{aligned}$$

The above derivation shows that if there exists a function H satisfying (17), then there exists a function J such that $af \cdot gh \ (n+1) \cdot N = J \ (agn \ n)$ holds. Function J is defined by

$$J \ x \ a = (r \ x \ a) \oplus ((\oplus / \cdot (H \ a \cdot (r' \ x) *) *) \ (rhss \ g \ a)) \quad (18)$$

It remains to prove that there exists a function H such that equation (17) is satisfied, and that function K defined in equation (13) is monotonic. The former condition is discussed in the next subsection. The latter condition is satisfied if operator \oplus is monotonic in both its arguments (which is for example true for the join \sqcup_c of the semilattice c by means of which the semilattice E is defined), and if H is monotonic in its second argument. From now on we assume $\oplus = \sqcup_c$.

We give an operational interpretation of the functions we have derived. Given a grammar g and a CPO (E, \sqsubseteq_E) , we compute the least fixed point of function K , starting with $K \perp$, where \perp is the bottom of E , and repeatedly applying K until we find a value x such that $K \ x = x$. Function K applies function J to all nonterminals of g . Function J takes the old value of K and a nonterminal nt , and returns the new value for nt by applying the function $H \ nt \cdot (r' \ x) *$ to all right-hand sides of the productions of nonterminal nt . The results are combined by taking the join \oplus of the values thus obtained, and, finally, by joining the result with the old value for nt .

5.3 The Conditions

In the previous subsections we have derived a function K by means of which a bottom-up grammar analysis problem can be solved. In the derivation we have imposed a number of conditions upon the components of the grammar analysis problems. This subsection discusses these conditions.

The first condition we imposed upon bottom-up grammar analysis problems is the following. We suppose there exists a join semilattice (c, \sqsubseteq_c) such that 1_\oplus is the bottom of c , and \oplus is the join \sqcup_c of c .

For the second condition we suppose that there exists a monotonic function H , such that equality (17) holds. Such a function H always exists, but the definition of a general function satisfying equality (17) is rather useless (and omitted): it recomputes the required information from scratch instead of using the available information, and is therefore highly inefficient. To obtain a practical solution for a bottom-up grammar analysis problem we discuss a special case in which we can find a monotonic function H that can be implemented as an efficient program.

Suppose the property function p is a catamorphism on *Rosetree*. Then we have that function $pn\ nt$ defined by $pn\ nt = qn\ nt \cdot exr*$, where function qn is the function of the *Rosetree* catamorphism for p , satisfies assumption (15). For the left-hand expression of equation (17) we have the following equality

$$\oplus / \cdot (pn\ nt)* \cdot cp \cdot ((top \Delta p)* \cdot (gh\ n))* = qn\ nt \cdot exr* \cdot (id \Delta (af \cdot gh\ n))*$$

provided there exists a function qn such that

$$p \cdot (Node\ nt) = qn\ nt \cdot p* \quad (19)$$

$$\oplus / \cdot (qn\ nt)* \cdot cp = qn\ nt \cdot (\oplus /)* \quad (20)$$

It follows that function H can be defined by $H\ nt = qn\ nt \cdot exr*$. The second assumption (20) is still rather unwieldy, and can be simplified. To obtain a simpler condition we apply the theory for cp developed in [5]. For that purpose, we first assume that function $qn\ nt$ is a reduction, that is, there exists an operator \otimes with unit 1_\otimes such that $qn\ nt = \otimes /$. Now we apply a theorem from [5], which states that (20) holds, provided the sections $(a\otimes)$ and $(\otimes a)$ distribute over operator \oplus , and provided for all y , $1_\oplus \otimes y = y \otimes 1_\oplus = 1_\oplus$. Function $H\ nt$ is monotonic provided function $qn\ nt$ is monotonic, and function $qn\ nt$ is monotonic provided operator \otimes is monotonic in both arguments.

5.4 Examples

This section shows how we apply the theory derived in the previous section to the examples of bottom-up grammar analysis problems given in Section 4. The algorithm derived in the previous section can be used to solve a bottom-up grammar analysis problem provided the components of the grammar analysis problem satisfy the conditions given in the previous section.

EMPTY

We verify the conditions the components of the definition of the bottom-up grammar analysis problem **EMPTY** have to satisfy.

First, the join semilattice (c, \sqsubseteq_c) upon which the join semilattice (E, \sqsubseteq_E) is built is the join semilattice of booleans.

For the second assumption, we have to construct a function H such that equation (17) holds. To obtain a definition of function H that can be implemented as an efficient program, we verify the conditions listed in the previous subsection. We have to show that function p defined by $p = ([] =) \cdot sen$ is a *Rosetree* catamorphism, i.e., there should exist a function qn such that $p\ (Node\ nt\ x) = qn\ nt\ (p*\ x)$ holds. Function qn is defined by $qn\ nt\ x = and\ x$. Furthermore, we have to show that \wedge , the operator of the reduction for *and*, distributes backwards and forwards over \vee , and that *false* is a zero of \wedge . These equalities hold for *false*, \vee and \wedge . Finally, \wedge is monotonic in both arguments.

FIRST

We verify the conditions the components of the definition of the bottom-up grammar analysis problem **FIRST** have to satisfy.

First the join semilattice (c, \sqsubseteq_c) upon which the join semilattice (E, \sqsubseteq_E) is built is the join semilattice of terminals, where c is the set of terminals, the relation \sqsubseteq_c is

the subset relation, $[]$ is the bottom of c , and the join \sqcup_c is set union, or $rmdups \cdot \sqcup$. Clearly, set union is associative, and $[]$ is the unit of set union.

For the second assumption, we have to construct a function H that can be implemented as an efficient program, such that equation (17) holds. The condition (20) given in the previous subsection does not hold for function p defined by $p = take\ 1 \cdot sen$. It is not difficult to find a Rosetree catamorphism for p , so (19) is satisfied, but the second requirement (20) does not hold. It follows that we have to find another way to construct function H . Function H is defined by $H\ nt = foldr\ t\ 1_\oplus$ where function t is defined as follows. If the current symbol in the right-hand side of a production is a terminal, then the symbols that can appear as the first symbol of a string are the symbols found until then, and no more: $t(T\ b, y)\ x = y$. If the current symbol in the right-hand side of a production is a nonterminal $N \cdot a$, then we distinguish two cases depending on whether or not $N\ a$ can derive the empty string. If $N\ a$ can derive the empty string, then the symbols that can appear as the first symbol of a string are the symbols found until then together with the first symbols of the remaining part of the production. If $N\ a$ cannot derive the empty string then the symbols that can appear as the first symbol of a string are the symbols found until then, and no more.

$$t(N\ a, y)\ x = \begin{cases} rmdups(y \sqcup x) & \text{if } at\ empties\ a \\ y & \text{otherwise} \end{cases}$$

We can prove equation (17) for function H thus defined by induction to the structure of lists: apply both sides to $[]$ and $[a] \sqcup x$, and show that the resulting expressions have the same recursive structure. It can be shown that H is monotonic.

6 Implementation

The definitions of some of the functions and datatypes given above are translated into Gofer as follows (the Gofer text is set in two columns).

```

split f g x = (f x , g x)           ,Eq [(a,[b])]
                                         ,Eq [[b]]
data Symbol a b = N a | T b          ,Semilattice [b]
                                         ) =>
                                         Grammar a b -> [(a,[b])]

type Grammar a b =                   firsts g =
  (a,[(a,[Symbol a b])])             ag
                                         g
                                         (\nt x -> foldr t bottom x)
                                         (\b -> [b])
                                         where
                                         t (N a,y) x
                                         | eg 'at' a = nub (y ++ x)
                                         | otherwise = y
rhss :: Eq a => Grammar a b -> [a]   t (T b,y) x = y
                                         eg = empties g
                                         nt : Eq a => Grammar a b -> a -> b
rhss g nt = [rhs | (z,rhs) <- snd g, z==nt]
  nts :: Eq a => Grammar a b -> [a]
  nts g = nub (map fst (snd g))
                                         at :: Eq a => [(a,b)] -> a -> b

```

```

class Semilattice a where
  join :: a -> a -> a
  bottom :: a

instance Semilattice Bool where
  join = (||)
  bottom = False
  .

instance Eq a => Semilattice [a]
  where
    join = \a b -> nub (a ++ b)
    bottom = []
  .

lfp :: Eq a => (a -> a) -> a -> a
lfp f x | x == f x = x
| otherwise = lfp f (f x)

lub :: Semilattice a => [a] -> a
lub = foldl join bottom

empties :: Eq [(a,Bool)] =>
  Grammar a b ->
  [(a,Bool)]
empties g =
  ag
  g
  (\nt x -> and (map snd x))
  (\a -> False)

firsts :: (Eq [(a,Bool)])

```

```

at xys x =
  head [y | (z,y) <- xys, z==x]

ag :: (Eq a
      ,Eq [(a,c)]
      ,Eq [c]
      ,Semilattice c
      ) =>
  Grammar a b ->
  (a -> [(Symbol a b,c)] -> c) ->
  (b -> c) ->
  [(a,c)]

ag g pn pl =
  lfp
  k
  (map
    (split id (\x -> bottom))
    nt's)
  where
    nt's = nts g
    k x = map (split id (j x)) nt's
    j x nt = (r x nt) `join` ((lub
      .map (pn nt
            .map (r' x)
            )
      .rhss g)
      nt
      )
  r x nt = at x nt
  r' x (N a) = (N a, r x a)
  r' x (T b) = (T b, pl b)

```

There are a number of ways in which a more efficient program can be obtained, for example by changing the definition of function `lfp`. For a discussion on this subject the reader is referred to [6].

7 Conclusions

This paper discusses bottom-up grammar analysis problems. We give a very general specification of bottom-up grammar analysis problems, and from this specification we derive, by means of program transformation applying laws to the components of the intermediate expressions, an algorithm for performing bottom-up grammar analysis. The driving force in the derivation of the algorithm is the construction of a fixed point. To obtain such a fixed point a number of conditions have to be imposed upon the components of the bottom-up grammar analysis problem. Thus we derive both the algorithm and the conditions under which the fixed point exists in one go. The derivation is an example of a derivation of a real-world program. The research reported on in this paper is still in progress: in the next version we want to split the calculation in two parts. The first part of the derivation assumes that

the function that computes the property of a parse tree is a *Rosetree* catamorphism and the second part of the derivation adds, if necessary, the extra information (for example in the case of *firsts*, where we use information about the *empties*). This simplifies the derivation. Future research will be directed towards the derivation of an algorithm for top-down grammar analysis.

Acknowledgements

Erik Meijer and Graham Hutton meticulously read an earlier version of this paper.

References

- [1] R.S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume F36 of *NATO ASI Series*, pages 5–42. Springer-Verlag, 1987.
- [2] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.
- [3] J.H. Fasel, P. Hudak, S. Peyton Jones, and P. Wadler. Sigplan Notices Special Issue on the Functional Programming Language Haskell. *ACM SIGPLAN notices*, 27(5), 1992.
- [4] Andy Gill and Simon Marlow. Happy manual. Published on comp.lang.functional, 1993.
- [5] J. Jeuring. *Theories for Algorithm Calculation*. PhD thesis, Utrecht University, 1993.
- [6] J. Jeuring and S.D. Swierstra. Bottom-up grammar analysis —a functional formulation—. Technical Report UU-CS-1994-01, Utrecht University, 1994.
- [7] M.P. Jones. Introduction to Gofer 2.20. Programming Research Group, Oxford University, 1992.
- [8] G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.
- [9] L. Meertens. Algorithmics—towards programming as a mathematical activity. In J.W. de Bakker, M. Hazewinkel, and J.K. Lenstra, editors, *Proceedings of the CWI Symposium on Mathematics and Computer Science*, volume 1 of *CWI Monographs*, pages 289–334. North-Holland, 1986.
- [10] Torben Mogensen. Ratatosk – a parser generator and scanner generator for Gofer. Published on comp.lang.functional, 1993.
- [11] Ulrich Möncke and Reinhard Wilhelm. Grammar flow analysis. In *Attribute Grammars, Applications and Systems, SAGA '91*, pages 151–186. Springer-Verlag, New York, 1991. LNCS 545.
- [12] Simon L. Peyton Jones. Yacc in Sasl – an exercise in functional programming. *Software–Practice and Experience*, 15(8):807–820, 1985.
- [13] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.

First-Class Polymorphism for ML

Stefan Kahrs^{*}

University of Edinburgh
Laboratory for Foundations of Computer Science
King's Buildings, EH9 3JZ

Abstract. Polymorphism in ML is implicit: type variables are silently introduced and eliminated. The lack of an explicit declaration of type variables restricts the expressiveness of parameterised modules (functors). Certain polymorphic functions cannot be expressed as functors, because implicit type parameters of polymorphic functions are in one respect more powerful than formal type parameters of functors.

The title suggests that this lack of expressiveness is due to a restricted ability to abstract — polymorphism is restricted. Type variables can only be abstracted from value declarations, but not from other forms of declarations, especially not from structure declarations.

The paper shows in the case of Standard ML how (syntax and) semantics can be modified to fill this language gap. This is not so much a question of programming language design as a contribution for better understanding the relationship between polymorphic functions, polymorphic types, and functors.

1 Introduction

Hindley-Milner polymorphism [6, 14] is the basis of the type systems of most modern functional programming languages. The computational, dynamic aspects of polymorphism are well-understood and the reader is assumed to be familiar with them. Less well-understood are the static aspects of polymorphism.

Polymorphism (for values) is introduced whenever type variables are abstracted in a *value* declaration. However, wide-spread functional languages like Standard ML [17] (SML for short) and Miranda² [8] have no corresponding introduction of polymorphism for *arbitrary* declarations, in particular polymorphism for modules is entirely missing. In practise, this is not a problem for languages which do not support parameterised modules, e.g. Haskell [9]. In the following we shall stick to SML, mainly because its formal definition [16] enables us to discuss the semantic issues involved; apart from that, this choice is not essential: the only essential ingredients for our approach are ML-style polymorphism and the presence of parameterised modules.

The lack of polymorphism for *structures* (modules in SML) can be illustrated by comparing the expressive powers of *functors* (parameterised modules in SML) and polymorphic functions.

^{*} The research reported here was supported by SERC grant GR/J 07303.

² Miranda is a trademark of Research Software Ltd.

```
fun foldleft (f,n) nil = n
| foldleft (f,n) (x::xs) = foldleft (f,f(x,n)) xs;
```

The function `foldleft` is polymorphic; the type of the parameter `(f,n)` is $(\alpha * \beta \rightarrow \beta) * \beta$ for arbitrary types α and β . One can think of α and β as implicit type parameters of `foldleft`. We can try to make this explicit by turning `foldleft` into a functor:

```
functor FOLDLEFT (type A; type B; val f: A*B->B; val n: B) =
  struct
    local fun loop n' nil = n'
           | loop n' (x::xs) = loop (f(x,n')) xs
    in    val foldleft = loop n
    end
  end;
```

The polymorphic function `foldleft` and the functor `FOLDLEFT` seem to have equal expressive power in the following sense: whenever we can write a value declaration that instantiates `foldleft` with its first argument, like

```
val foldinstance = foldleft exp;
```

then there is an equivalent instantiation of the functor `FOLDLEFT`:

```
local structure Aux =
  FOLDLEFT(type A = ... ; type B = ...
           val (f,n) = exp)
in   val foldinstance = Aux.foldleft
end;
```

The only problems here are the ellipses which have to be replaced by appropriate type expressions. Since these types can be inferred from `exp`, this does not seem to be a real problem. However, there is one, and we can observe it when considering the standard example of instantiation of `foldleft`, list reversal:

```
val reverse = foldleft (op ::,nil);
```

This is a *polymorphic* instantiation of `foldleft`: the defined function `reverse` is itself polymorphic. Hence, one might expect the appropriate type declarations in the corresponding instantiation of `FOLDLEFT` to be: `type A = 'a` and `type B = 'a list`. But it is unclear where the type variable '`'a`' comes from, and indeed type declarations of this form are illegal in SML (and Miranda) — type variables on the right-hand side of a type declaration have to be introduced on the left-hand side. The reason for this restriction is partly a concern about soundness, and partly lack of imagination (or desire for simplicity) on the side of the language designers.

I will show how this restriction can (safely) be eliminated, i.e. in what way the syntax and semantics of SML can be modified to support polymorphism for structures and functors.

2 Polymorphism vs. Functors

Parameterised modules in SML are called *functors*. A functor maps structures that match its interface (a *signature*) to structures. SML functors are similar to parameterised “scripts” in Miranda [8] and parameterised “modules” in ET [3].

SML functors generalise polymorphic functions in the following sense. Parameters of polymorphic functions are types (implicit) and values (explicit). Functor parameters can be (polymorphic) values, (parameterised) types, and structures. Therefore, a functor can be supplied with the arguments of a polymorphic function via its input signature. The example of `foldleft` and `FOLDLEFT` (motivated by section 9.3 in [18]) illustrates how a functor can simulate a polymorphic function, but also in which way this simulation is restricted. Apart from that, functors are *more* powerful than polymorphic functions, because the value parameters can themselves be polymorphic and because the type parameters can be parametric. Here is an example (adapted from [19]) that shows the extra power of functors.

```

signature MONAD =
sig
  type 'a M
  val unitM: 'a -> 'a M
  val bindM: 'a M -> ('a -> 'b M) -> 'b M
end;
functor Monad(include MONAD) =
struct
  fun mapM f m = bindM m (unitM o f)
  fun joinM z = bindM z (fn x => x)
end;

```

The functor `Monad` and the signature `MONAD` both use the formal parameterised type associated with `M` with different arguments, in particular `joinM` has type $('a M) M \rightarrow 'a M$. Analogously, the polymorphic function `bindM` is used with different type instances in the body of the functor. This cannot be expressed with polymorphic functions; $\lambda\omega$ is the weakest type system in Barendregt’s λ -cube [1] that can express the `Monad` functor.

Our goal is to make functors properly more powerful than polymorphic functions — not so much because this lacking power is badly missed in programming practice, but in pursuit of a better understanding of the two concepts. We shall first look at ways to circumvent the restriction of polymorphic functor instantiations in the existing language.

It is still possible to define a uniform list reversal as an instance of `FOLDLEFT`, but not as a polymorphic *function* — we can define it as a *functor*:

```

functor REVERSE(type T) = FOLDLEFT(
  type A = T; type B = T list
  val f = op :: ; val n = nil);

```

Syntactic restrictions for functor instantiation make the usage of the functor **REVERSE** slightly awkward — it is not possible to instantiate a functor within an expression, for example as in **REVERSE(type T=int).foldleft[1,2,3]**; the functor instantiation has first to be assigned to a structure name via a structure declaration. Again this means that the polymorphic use of **REVERSE** within the definition of a polymorphic function is not possible, unless the polymorphic function is turned into a functor. In Miranda, the analogous situation is even worse, because each parameterised module is a file.

It should be emphasised that these restrictions on functor usage are not merely syntactical accidents; they are deliberate design decisions in the concerned languages to keep all type-checking at compile-time. See [4] for a discussion on this *phase distinction*.

The problem of polymorphic instantiations of a functor has been noticed before, for instance by Hinze in [7]. The solution suggested there is to make formal type parameters of the functor parametric. In our **FOLDLEFT** example, this concerns its type parameters **A** and **B**; with the so-modified functor we can define a polymorphic **reverse** by functor instantiation³.

```
functor FOLDLEFT_1 (type 'a A; type 'b B;
                     val f: 'a A * 'a B -> 'a B; val n: 'b B) =
  struct local fun loop r nil = r
              | loop r (x::xs) = loop (f(x,r)) xs
            in   val foldleft = loop n
            end
  end;
local structure Aux = FOLDLEFT_1(type 'a A = 'a;
                                  type 'b B = 'b list;
                                  val f = op ::;
                                  val n = nil)
in   val reverse = Aux.foldleft : 'a list -> 'a list
end;
```

The new functor **FOLDLEFT_1** is not restricted to non-monomorphic applications, because type declarations like **type 'a A = int** that erase a type argument are legal in SML. Indeed, **FOLDLEFT** can be expressed as an instance of **FOLDLEFT_1**. But this approach has two snags. The type parameters for **A** and **B** have nothing to do with the functor itself: neither its interface nor its body apply **A** or **B** to anything different from a type variable. From a methodological point of view, the functor interface is therefore a bad place for introducing these type parameters. More importantly, we have not really solved yet the problem of how to get a polymorphic function, which is an instance of **foldleft**, by instantiating the corresponding functor — only the special case where polymorphism is restricted to *at most one* type parameter. For example, we cannot define the (fully) polymorphic function **map** by instantiating **FOLDLEFT_1**, because **map** is parametric in two type variables. Of course, we can again abstract a further variable and

³ Some SML implementations struggle with this example, but it is perfectly legal.

define another functor **FOLDLEFT_2**, but each abstraction step makes the syntax of the functor (and its monomorphic instantiations) increasingly messy without solving the general problem.

We can observe the limitations of functor polymorphism more clearly in the **Monad** example. One of the classic examples for monads are continuation monads (also adapted from [19]):

```
structure Contin: MONAD =
  struct
    type 'a M = ('a -> Answer) -> Answer
    fun unitM a = fn c => c a
    fun bindM m k = fn c => m (fn a => k a c)
  end;
structure ContMon = Monad(open Contin);
```

Wadler suggests various choices for type **Answer**. We can express this in SML by turning the above piece of code into a functor with parameter **Answer**. However, this would enforce a new instance of **Monad** for every choice for **Answer** which is a bit of a waste. Instead, it would be more natural to replace **Answer** by a free type variable '**b**', giving us a polymorphic continuation monad.

3 Polymorphism in SML

The example of **foldleft** shows how polymorphism is usually treated in SML and many related languages. The polymorphism of **foldleft** has been silently introduced; we can see this more clearly by mixing the explicit polymorphism of the type system $\lambda 2$ with ML code:

```
val rec foldleft =  $\lambda\alpha : * . \lambda\beta : * .$ 
  fn (f: $\alpha \times \beta \rightarrow \beta$ ,n: $\beta$ ) => fn ls:  $\alpha$  list =>
  case ls of nil  $\alpha$  => n
  | (op ::)  $\alpha$  (x, xs) => foldleft  $\alpha$   $\beta$  (f,f(x,n)) xs
```

The $*$ is the universe of types, i.e. $\lambda\alpha : *$ denotes type variable abstraction in $\lambda 2$, see [1]. Instantiation of polymorphic values with types (application $t\tau$ of terms t to types τ in $\lambda 2$) is implicit in ML; similarly the introduction of types with variables like α for value expressions. On the level of types, application and abstraction of types are always explicit, for example in the declaration of type **list**:

```
datatype 'a list = nil | :: of 'a * 'a list
```

Here we have an explicit type variable '**a**', its explicit abstraction on the left-hand side and an explicit type application, '**a list**' on the right-hand side.

This syntactic difference in polymorphism has a semantic equivalent — SML uses two different notions of type variable abstraction for values and types, called *type schemes* and *type functions*. The type function $\Lambda\alpha.\alpha t$ is the semantic value

of `list` (t is a type name, a kind of personal identification number for a type), the type scheme $\forall\alpha.\alpha t$ is the static semantic value of `nil`; type names have an arity and (constructed) types are formed by applying an n -ary type name to n types, type name application being written postfix. The difference between type functions and type schemes is that instantiation of bound type variables is always explicit for type functions and always implicit for type schemes. The semantic reason for distinguishing these two forms of abstraction is certain equivalences that apply to type schemes: for example, the type schemes $\forall\alpha\forall\beta.\tau$ and $\forall\beta\forall\alpha.\tau$ are equal, but the corresponding type functions $\Lambda\alpha\Lambda\beta.\tau$ and $\Lambda\beta\Lambda\alpha.\tau$ are different, provided α or β occurs in τ .

There is another important difference between polymorphism for values and polymorphism for types in SML: the effect of nested declarations. Instead of defining `foldleft` by direct recursion, we can exploit the fact that `f` is fixed throughout the recursion (cf. `FOLDLEFT` above):

```
fun foldleft (f,n) ls =
  let fun loop r nil = r
      | loop r (x::xs) = loop (f(x,r)) xs
    in  loop n ls
  end;
```

The local function `loop` is monomorphic, it does not abstract type variables. We can see this by annotating the ML code with type abstractions and applications (exercise for the reader). However, the declaration of `loop` still contains (implicitly) a free type variable α for the instantiation of the list constructors `nil` and `::`. This type variable is introduced *in the context* of the declaration of `loop`.

Concerning type declarations, the rôle of nested declarations is different. Although a type declaration can occur in a context which contains free type variables (in SML; not in Miranda), all type variables occurring on its right-hand side *must* be declared on its left-hand side, they cannot come from the context. The language definition of SML imposes this restriction.

We want to lift this syntactic restriction, because it prevents us from writing the polymorphic functor instantiations. This raises the question how free type variables are introduced and eliminated, and in particular what the semantic equivalent of the elimination operation is.

4 Type Variable Declarations

Before we consider the introduction and elimination of type variables for arbitrary declarations, let us look at the semantic rule⁴ that defines the corresponding operation for value declarations, i.e. that introduces type schemes for value variables — rule 17 in the definition of the static semantics of SML [16]:

⁴ The rule presented here is a simplified version — I have removed the parts that deal with imperative type variables, as they have no particular significance in this context.

$$\frac{C + U \vdash valbind \Rightarrow VE \quad VE' = Clos_C VE}{C \vdash \mathbf{val}_U valbind \Rightarrow VE' \text{ in } Env}$$

The non-terminal *valbind* stands for a value declaration. Sentences of the semantics of SML of the form $C \vdash phrase \Rightarrow VE$ can be read as: in the context C the syntactical phrase *phrase* gives rise to (*elaborates to* is the technical term) a variable environment VE . Variable environments bind identifiers to their type schemes; they also occur as components of general environments ($E \in Env$) that can contain other bindings as well. “ VE' in Env ” is a general environment containing the variable environment VE' but no other bindings.

$C + U$ is a context in which type variables from U are declared to be free and $Clos_C VE$ is a variable environment obtained from VE by abstracting all type variables not free in C ; this abstraction introduces type schemes, pointwise for each variable bound in VE . It may be a bit surprising that the result of a type variable abstraction from a variable environment is not a mapping from types to variable environments, but another variable environment. The justification goes as follows: a (static) variable environment can be seen as a tuple of types (or rather type schemes), indexed by the bound identifiers. If we extend the type system $\lambda 2$ with binary products, the following types are “isomorphic”:

$$\Pi\alpha : *. (T(\alpha) \times U(\alpha)) \cong (\Pi\alpha : *. T(\alpha)) \times (\Pi\beta : *. U(\beta))$$

The abstraction on the left-hand side abstracts a type variable from a tuple, the right-hand side is a tuple (type) of abstractions. For instance, the mapping from left to right can be given as the expression $\lambda x : P. (\lambda\alpha : *. \pi_1(x\alpha), \lambda\beta : *. \pi_2(x\beta))$ in $\lambda 2$, where P is the type on the left-hand side of \cong .

I write “isomorphic” in quotes, because they are only isomorphic in a weak sense as indicated by Di Cosmo in [2]: we have to impose a few equivalences to make the composition of both mappings equal to the identity. These equivalences are $=_{\beta\eta}$, surjective pairing, and the equation $(fx, gx) = (f, g)x$. This weak isomorphism is the justification (in SML) for performing the abstraction pointwise, i.e. for picking the type on the right.

Coming back to the mentioned semantic rule in SML, it contains another slightly mysterious bit. The subscript U in $\mathbf{val}_U valbind$ is the set of type variables *scoped at this value declaration*. In other words: the rule incorporates introduction and elimination of free type variables. For value polymorphism, the scoping of type variables is a minor issue as it deals only with type variables that occur explicitly in the text while type scheme polymorphism is mainly tacit and operates on implicit type variables. Explicit type variables are not introduced by an explicit declaration, but rather attached to a value declaration by a general principle. Allowing free type variables to occur in other forms of declarations raises the need for an explicit form of type variable declaration. For the purposes of this paper, I suggest **typevars** *tyvarseq* as an additional form of declaration. The corresponding rule in the static semantics is quite simple:

$$\frac{}{C \vdash \mathbf{typevars} tyvarseq \Rightarrow [tyvarseq] \text{ in } Env}$$

4.1 Abstraction: General Idea

Type variables are not bound to anything⁵, so an environment simply contains a sequence of non-empty sequences of type variables, listing the free type variables in that environment. We shall see later why the semantic value of a type variable declaration is a *sequence of sequences* rather than a *set* of type variables. Semantically more interesting than introduction is elimination of free type variables. It seems natural to let type variable declarations follow the usual scoping rules for declarations and eliminate them at the end of their scope, for example by modifying the rule for local declarations accordingly:

$$\frac{C \vdash dec_1 \Rightarrow E_1 \quad C \oplus E_1 \vdash dec_2 \Rightarrow E_2}{C \vdash \text{local } dec_1 \text{ in } dec_2 \text{ end} \Rightarrow \text{abstract}_C(U \text{ of } E_1)(E_2)}$$

A local declaration `local dec1 in dec2 end` declares whatever E_2 declares; the declarations in dec_1 are auxiliary for dec_2 and their scope ends at the keyword `end`. Thus, if dec_1 contains type variable declarations their scope also ends at the end of the local declaration.

The difference from SML's rule for local declaration is the application of the abstraction operator to E_2 in the conclusion, rather than taking E_2 itself as the result. It would follow more closely the style of the SML definition if the abstraction were expressed as $\text{Clos}_C E_2$, but there is a problem: the operation Clos_C abstracts all type variables which are free in its argument and not free in C , but the order of abstraction is significant for type functions and hence for our generalised abstraction as well. I postpone the definition of *abstract*, until it is clearer which properties it should have. Consider an example that uses the feature of type variable declarations:

```
local typevars 'a
in    datatype list = nil | :: of 'a * list
      val foldleft = fn (f,n) =>
          let fun loop r nil = r
              | loop r (x::xs) = loop (f(x,r)) xs
          in loop n
          end
end;
```

Notice that the type variable '`'a`' in the declaration of `list` does *not* come from the left-hand side but from the context of the declaration. Within the local declaration, `list` is a monomorphic type and `foldleft` is only polymorphic in one type variable, its result type.

For merely pragmatic reasons, it is desirable to abstract type variables pointwise from the components of an environment. The components which matter in this respect are (i) type schemes, the semantic values of value variables, (ii) type functions, the semantic values of type constructors, and (iii) type names, the personal identification numbers of newly introduced constructed types.

⁵ One can think of a type variable as an ordinary variable bound to `*`, as in $\lambda 2$.

4.2 Abstraction: Gory Details

We would certainly like the abstraction operator to behave on type schemes just as Clos_C does, i.e. to increase the set of abstracted type variables of a type scheme: type abstraction introduces polymorphism. This principle already allows us to formulate `reverse` as an instance of `FOLDLEFT`:

```
local
  typevars 'a
  structure Aux =
    FOLDLEFT(type A = 'a
              type B = 'a list
              val f = op ::)
              val n = nil)
in  val reverse = Aux.foldleft
end
```

The only item that is subject to type variable abstraction in this example is the type of `reverse`, because its declaration is the only non-local one. Within the local declaration, `reverse` has the monomorphic type $'a \text{ list} \rightarrow 'a \text{ list}$, but $'a$ can be abstracted at the end of the local declaration, introducing a polymorphic `reverse`.

Abstracting a type variable from a *type name* increases the arity of that type name: the arity of `list` inside the `local` declaration is 0, but it is 1 outside, as we want be able to instantiate $'a$ with various types and as we have to distinguish these different instantiations semantically to preserve the soundness of the type system. We only need to abstract type variables on which a type name depends; in the example, `list` depends on $'a$, because $'a$ occurs freely in its constructor environment. For simplicity, we can assume that all (new) type names depend on all abstracted type variables.

The change of arity of a type name slightly complicates abstraction in the variable environment case, as further components of the environment may contain that type name and are thus affected by such a change: an environment is like a *dependent* n-tuple and we need weak isomorphisms operating on dependent tuples (see [13] for an introduction to Σ -types), in the binary case as follows:

$$\Pi\alpha : *. \Sigma x : T(\alpha). U(\alpha, x) \cong \Sigma x : (\Pi\alpha : *. T(\alpha)). \Pi\beta : *. U(\beta, x\beta)$$

Notice that x on the left-hand and right-hand side of the \cong has different arities; this corresponds to the different arities of a type name. Fortunately, this weak isomorphism exists and is similarly straightforward as in the non-dependent case.

One case remains: type variable abstraction for type functions. We could stick to the principle that type abstraction always introduces *implicit* polymorphism, so that a type function may have explicit *and* implicit parameters. In the example it would mean that `list` has an *implicit* parameter outside the `local` declaration and that it would be the task of type inference to compute it, similarly as it computes the implicit type parameter of `nil`. This is surely possible but

seems rather unusual and involves a number of language design problems, e.g. its interaction with the module system or whether it would be possible to restrict implicit polymorphism in type expressions. I shall not pursue this approach here.

The alternative is to turn abstracted type variables into *explicit* parameters of a type function. Since these additional parameters are explicit, there is a corresponding effect on the syntactical level: the arities of *type constructors* change as well. In the example, the type constructor `list` has arity 1 outside the `local` declaration: it requires an argument when used in type expressions.

To be able to change the arity of local type names, it would be useful to explicitly keep track of local type names as an additional component of environments. Implementations do that anyway, and some arguments why the SML definition should also be explicit about it can be found in section 9.2 of [10].

SML already provides a mechanism to replace type names: these are the so-called *realisations* which are used for structure/signature matching. Basically, a realisation is a finite map from k -ary type names to k -ary type functions; it can be applied to various semantic objects by replacing the type names throughout the object. Realisation application can be seen as second-order substitution.

For the abstraction operation, such functions are more complicated, i.e. we need more structure for realisations, type functions etc.

- Analogously to type declarations, type functions can now contain free type variables.
- The arity of a type name (or a type function) in SML is a natural number n ; it is convenient to generalise this to a sequence of natural numbers n^* , which notationally supports curried application of type constructors.
- A realisation in SML maps a k -ary type name to a k -ary type function; here, if we abstract a sequence α^* of non-empty sequences of type variables from an environment, the corresponding abstraction realisation maps k^* -ary type names to $n^* \cdot k^*$ -ary type functions, where \cdot is list concatenation and $n^* = \text{map length } \alpha^*$.

Let C be a fixed context and α^* be a fixed sequence of non-empty sequences of type variables and $n^* = \text{map length } \alpha^*$. An *abstraction realisation* φ is an injective map from k^* -ary type names (not in C) to $n^* \cdot k^*$ -ary type names (also not in C). We need injectivity and disjointness from the type names in C in the result to preserve the soundness of the type system⁶. Because the arity of type names is not preserved, we have to redefine the application of abstraction realisations to constructed types:

$$\begin{aligned} t \in \text{Dom } \varphi &\Rightarrow \varphi(\tau^* \cdot t) = (\alpha^* \cdot \varphi(\tau^*)) \cdot \varphi(t) \\ t \notin \text{Dom } \varphi &\Rightarrow \varphi(\tau^* \cdot t) = \varphi(\tau^*) \cdot t \end{aligned}$$

where τ^* is a sequence of non-empty sequences of types and t is a type name. Furthermore the arity of type function changes, i.e. $\varphi(\Lambda\beta^*.t) = \Lambda\alpha^*\beta^*. \varphi(t)$.

⁶ A technical remark: unfortunately, the *consistency* condition for semantic objects is too weak for this purpose.

Now we can define $\text{abstract}_C(\alpha^*)(E)$ as $\text{Clos}_C(\varphi E)$ where φ is an arbitrary abstraction realisation w.r.t. context C and type variables α^* , which is defined on all type names in E that are not in C. The closure operator Clos_C introduces type schemes in variable (and constructor) environments.

From the language design point of view, the abstraction operator has another merit. It allows to separate type abstractions from datatype declarations. There are some reasons to enforce this separation as the only form for recursive type declarations. The static semantic rules for value declarations use the same structure, i.e. type variable abstraction is imposed after the recursion has been solved. Because of this, it is not possible to define structurally inductive functions (which pass SML's type-check) for certain recursive types.

Strengthening the static semantic rules for value declarations such that structurally inductive functions for all recursive datatypes are typable makes *typability* [1] undecidable, because this is equivalent to solving arbitrary semi-unification problems [11]. Although “undecidable” surprisingly does not imply “impractical” in this case (see [5]), a type-checker that is necessarily non-terminating for some inputs may make some people feel uncomfortable. Having abstraction and datatype declaration as two separate concepts allows to impose the same restrictions on recursive types as on recursive functions, such that any recursive type has its corresponding recursive functions and vice versa.

5 Imperative Features

Naive polymorphism is unsound in connection with certain imperative features, for example it is unsound to have updatable polymorphic variables (*references*). For this reason, SML has a second form of type variables, *imperative type variables*. Abstraction from imperative type variables is restricted to so-called non-expansive objects; non-expansiveness is a sufficient condition for preventing polymorphic references and exceptions.

Standard ML implements this restriction by modifying its abstraction operator Clos and making $\text{Clos}_C VE$ dependent on whether VE was derived from an expansive (value) declaration or not. A similar modification would be necessary for our generalised abstraction operator. For example, we could regard a sequence of declarations (a structure) as expansive if any of its elementary declarations is expansive; the expansive elementary declarations consist of exception declarations and expansive value declarations. An attempt to abstract imperative type variables from an environment which was derived from an expansive structure could then be regarded as an error.

But this is not the only difficulty. Since we can now also abstract type variables from type constructors, we have additional problems: the abstraction may put *applicative* type variables (unrestricted polymorphism) into places where they should not be, example:

```
functor EXCEPT(type t) =
  struct exception A of t end;
structure S = let typevars 'a in EXCEPT(type t = 'a) end;
```

The component type of an exception is not allowed to contain applicative type variables (for soundness reasons; see [15], page 42), but in the above example abstraction and module instantiation unfortunately outwit this restriction.

The straightforward solution is to supply type names with an additional “imperative” attribute (similar to the equality attribute) and to require realisations to assign only imperative types to imperative type names. The example would be ruled out, as `t` is imperative and `'a` is not. Even with imperative type variables there are problems here, because functor bodies can be expansive, as the example shows. One possible way of solving this problem is to require that formal imperative type names are mapped by a realisation to closed imperative types; in other words: to essentially restrict generalised type abstraction to the applicative case.

The presence of non-imperative type names has a number of other effects on the language.

- A type $\tau^* t$ is imperative if t is an imperative type name and all types in τ^* are imperative. This implies a restriction on specialisation of imperative polymorphism and the types of exception constructors.
- We have to compute an imperative attribute for newly introduced datatypes. This is completely analogous to the equality attribute, i.e. a new datatype is imperative if (roughly) all its constructors have imperative types as argument types.
- We need a feature to specify the imperative attribute of a formal type parameter of a functor, analogous to `eqtype` for the specification of the equality attribute. To keep the extension upwards-compatible with the existing language, we can take a type specification `type ty` to specify that `ty` is imperative and add a feature for the specification of (possibly) non-imperative types.

A nice side-effect of this approach is that the attributes for equality and imperativeness are treated completely analogous, i.e. they are attributes of type names and type variables. This fits very well with a Haskell-like understanding of these attributes as type classes.

Instead of this rather sophisticated approach to imperative features, one could instead employ the method suggested by Leroy in Chapter 6 of his thesis [12]. It can be roughly described by the slogan: “type variable abstraction introduces closures”, i.e. (even implicit) type application forces re-evaluation. This method eliminates all soundness problems with imperative features, including the ones mentioned in this section.

6 Related Module Systems

In the following, we briefly discuss the (potential) rôle of type variable abstraction in the module systems of ET [3] and Miranda [8, 7].

6.1 ET

The (functional logic) programming language ET is the only programming language with a notion of polymorphic abstraction for types I am aware of. Because ET's module system is flat (no substructures) and because type declarations are only permitted at top level, ET has only one place for free type variables — instantiation of functors. Type variables can be locally free for a functor instantiation. In ML syntax, ET's functor instantiations all have the following shape:

```
local structure Aux =
  let typevars 'a
    in FOLDLEFT(type A = 'a; type B = 'a list;
                  val f = op ::; val n = nil)
    end;
in   val reverse = Aux.foldleft
end
```

The semantic operations that support such a notation in ET differ slightly from the method described above for SML. Type declarations in the functor argument are required to be non-parametric, i.e. the corresponding type functions are of arity 0. For the functor body, these types are treated like free type variables and are abstracted from the exported objects. A functor instance (with or without free type variables) instantiates these variables with actual types and restores the old arity of all type constructors, before all free type variables of the functor instance are abstracted.

The difference to the method described for SML is in sharing of types. The first abstraction does not depend on the module instance and is performed only once — an ET functor contains free type names, in contrast to SML where each functor instance makes a fresh copy of these type names. Thus, types obtained from different functor instances in ET are compatible and an ET functor application never generates new datatypes. In other words: ET functors are extensional — applied to the same arguments they deliver the same results.

6.2 Miranda

Miranda's module system has a similar structure to ET: modules are flat and types only exist on top level, but it does not support type variable abstraction from functor instances. Concerning type compatibility of functor instances, Miranda follows SML. Since the specification of parameterised types in functor interfaces is possible, ET's approach is not feasible anyway, because this would require third order type constructors and second order type variables, making type inference undecidable.

Miranda supports recursive functor instantiation, i.e. the exported objects of a functor can be used to provide it with its input parameters. It is possible to create recursive types by recursive functor instantiation. For the abstraction of type variables, here we have again the problem whether abstraction and recursion

take place simultaneously, or whether abstraction is imposed *after* the recursion. An example (in ML-style syntax) should make it clear:

```
functor COPY (type t) =
  struct
    type u = t
  end;
structure rec C =
  let typevars 'a
  in  COPY(datatype t = nil | cons of 'a * 'a C.u)
  end;
```

The functor `COPY` exports the non-parameterised type `u`, which abstraction makes `u` parametric *before* it is stored in `C`, i.e. `C.u` is parametric and has to be provided with an argument even in recursive occurrences. One can also express abstraction *after* recursion by moving the declaration of `'a` outside:

```
local typevars 'a
in   structure rec C =
      COPY(datatype t = nil | cons of 'a * C.u)
end;
```

In this slight modification of the last example, `C.u` is non-parametric within the recursion, but becomes a parametric type at the end of the `local`.

7 Conclusion

Polymorphism in languages like Standard ML or Miranda is restricted because it allows to abstract type variables from declarations of values but not from declarations of types or structures. In this sense, polymorphism is not first-class. This restriction can be felt in the presence of parameterised modules, which in a strange way happen to be less expressive than polymorphic functions.

The syntactic cure is simple and — important from a language design point of view — easy to understand: introduce a new form of declaration, the explicit declaration of type variables. Such type variables are abstracted at the end of their scope. The semantic cure is a little bit more subtle, because type variable abstraction has to be defined for all possible environment components.

Such pointwise type variable abstraction from environment components is justified, for one can consider the global abstraction of types from a tuple to be isomorphic to the tuple with pointwise abstracted types. This isomorphism has already been exploited in the semantics of Standard ML and Miranda, but it becomes slightly more complicated in this setting as (general) environments have to be regarded as *dependent* tuples.

Acknowledgements

I would like to thank Bernd Gersdorf, Claudio Russo, Don Sannella and Andrzej Tarlecki and the ESOP referees for valuable discussions on this subject and feedback on an earlier version of this paper.

References

1. Hendrik P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science, Vol.2*, pages 117–309. Oxford Science Publications, 1992.
2. Roberto di Cosmo. Type isomorphisms in a type-assignment framework. In *19th ACM Symposium on Principles of Programming Languages*, pages 200–210, 1992.
3. Bernd Gersdorf. *Entwurf, formale Definition und Implementierung der funktional-logischen Programmiersprache ET*. PhD thesis, Universität Bremen, 1992. (mainly in German).
4. Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *17th ACM Symposium on Principles of Programming Languages*, pages 341–354, 1990.
5. Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, 1993.
6. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of AMS*, 146:29–60, 1969.
7. Ralf Hinze. *Einführung in die funktionale Programmierung mit Miranda*. Teubner, 1992. (in German).
8. Ian Holyer. *Functional Programming with Miranda*. Pitman, 1991.
9. P. Hudak, S. Peyton Jones, and P. Wadler. Report on the Programming Language Haskell, a Non-strict, Purely Functional Language. Technical report, University of Glasgow, 1992. (also in SIGPLAN Notices 27(5), May 1992).
10. Stefan Kahrs. Mistakes and ambiguities in the definition of Standard ML. Technical Report ECS-LFCS-93-257, University of Edinburgh, 1993.
11. A.J. Kfoury, J. Tiuryn, and P. Urcyczyn. The undecidability of the semi-unification problem. *Information and Computation*, 102(1):83–101, January 1993.
12. Xavier Leroy. Polymorphic typing of an algorithmic language. *Rapports de Recherche* No. 1778, INRIA, 1992.
13. Per Martin-Löf. An intuitionistic theory of types: predicative part. In Rose and Shepherdson, editors, *Logic Colloquium 1973*, pages 73–118. North-Holland, 1975.
14. Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
15. Robin Milner and Mads Tofte. *Commentary on Standard ML*. MIT Press, 1991.
16. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.
17. Laurence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.
18. Stefan Sokolowski. *Applicative High Order Programming*. Chapman & Hall Computing, 1991.
19. Philip Wadler. The essence of functional programming. In *19th ACM Symposium on Principles of Programming Languages*, pages 1–14, 1992.

Dimension Types

Andrew Kennedy

University of Cambridge
Computer Laboratory
Pembroke Street
Cambridge CB2 3QG
United Kingdom

`Andrew.Kennedy@ccl.cam.ac.uk`

Abstract. Scientists and engineers must ensure that physical equations are dimensionally consistent, but existing programming languages treat all numeric values as dimensionless. This paper extends a strongly-typed programming language with a notion of dimension type. Our approach improves on previous proposals in that dimension types may be polymorphic. Furthermore, any expression which is typable in the system has a most general type, and we describe an algorithm which infers this type automatically. The algorithm exploits equational unification over Abelian groups in addition to ordinary term unification. An implementation of the type system is described, extending the ML Kit compiler. Finally, we discuss the problem of obtaining a canonical form for principal types and sketch some more powerful systems which use dependent and higher-order polymorphic types.

1 Introduction

One aim behind strongly-typed languages is the detection of common programming errors before run-time. Types act as a constraint on the range of allowable expressions and stop ‘impossibilities’ happening when a program is run, such as the addition of an integer and a string.

In a similar way, scientists and engineers know that an equation cannot be correct if constraints on *dimensions* are broken. One can never add or subtract two values of differing dimension, and the multiplication or division of two values results in values whose dimensions are also multiplied or divided. Thus the sum of values with dimensions speed and time is a dimension error, whereas their product has dimension distance.

The addition of dimensions to a programming language has been suggested many times [KL78, Hou83, Geh85, Män86, DMM86, Bal87]. Some of this work is seriously flawed and most systems severely restrict the kind of programs that can be written. House’s extension to Pascal is much better [Hou83]. In a monomorphic language it allows functions to be polymorphic over the dimension of arguments. Since the submission of this paper an anonymous referee has pointed out work by Wand and O’Keefe on dimensional inference in the style of ML type inference [WO91]. In some ways this is similar to the approach taken here and a comparison with their system is presented later in this paper.

2 Some issues

2.1 Dimension, Unit and Representation

There is often confusion between the concepts of *dimension* and *unit* [Man87]. Two quantities with the same *dimension* describe the same kind of property, be it length, mass, force, or whatever. Two quantities with different *units* but the same dimension differ only by a scaling factor. A value measured in inches is 12 times the same value measured in feet—but both have the dimension *length*. We say that the two units are *commensurate* [KL78, DMM86]. These units have simple scaling conversions. More complicated are units such as temperature measured in degrees Celsius or Fahrenheit, and even worse, amplitude level in decibels.

Base dimensions are those which cannot be defined in terms of other dimensions. The International System of Units (SI) defines seven of these—length, mass, time, electric current, thermodynamic temperature, amount of substance and luminous intensity. *Derived* dimensions are defined in terms of existing dimensions, for example, acceleration is distance divided by time squared. Dimensions are conventionally written in an algebraic form inside square brackets [Lan51], so for example the dimensions of force are written [MLT^{-2}].

Similarly there are base units—the SI base dimensions just listed have respective units metres, kilograms, seconds, Amperes, Kelvin, moles and Candela. Examples of derived units include inches (0.0254 metres) and newtons (kgms^{-2}).

There is also the issue of *representation*: which numeric *type* is used to store the numeric value of the quantity in question. Electrical quantities are often represented using complex numbers, whereas for distance reals are more common, and for either of these many languages provide more than one level of precision.

Dimensionless quantities are common in science. Examples include refractive index, coefficient of restitution, angle and solid angle. The last two should properly be considered dimensionless though it is tempting to think otherwise. An angle is the ratio of two lengths (distance along an arc divided by the radius) and a solid angle is the ratio of two areas (surface area on a sphere divided by the square of the radius).

2.2 Types and Polymorphism

How do these concepts of dimension, unit, and representation fit with the conventional programming language notion of *type*?

Expressions in a strongly typed language must be *well-typed* to be acceptable to a compiler. In functional languages, for example, the rule for function application insists that an expression $e_1 e_2$ has type τ_2 if e_1 has an arrow type of the form $\tau_1 \rightarrow \tau_2$ and the argument e_2 has type τ_1 .

In a similar way, mathematical expressions must be *dimensionally consistent*. Expressions of the form $e_1 + e_2$ or $e_1 - e_2$ must have sub-expressions e_1 and e_2 of identical dimension. But in $e_1 e_2$ (product) the sub-expressions may have any dimension, say δ_1 and δ_2 giving a resultant dimension for the whole expression of $\delta_1 \delta_2$.

So it appears that dimensions can be treated as special kind of type in a programming context. But there is the question of what to do about representation. Do we associate particular dimensions with fixed numeric types (so current is always represented by a complex number, distance by a real), or do we parameterise numeric types on dimension and give the programmer the flexibility of choosing different representations for different quantities with the same dimension?

A *monomorphic* dimension type system is of limited value. For non-trivial programs we would like to write general-purpose functions which work over a range of dimensions. Even something as simple as a squaring function cannot be expressed in a monomorphic system. A modern polymorphic language would use quantified variables to express the idea that this function squares the dimension of its argument, for *any* dimension.

2.3 Type inference

Type systems such as that of Standard ML are designed so that the compiler can *infer* types if the programmer leaves them out. It turns out that this is possible for a dimension type system too.

A desirable property of inferred types is that they are the *most general* type, sometimes called *principal*. Any other valid typing can be obtained from this most general type by simple substitution for type variables. Our system does have this feature, and an algorithm is described which finds the principal type if one exists.

3 The idea

The system described here is in the spirit of ML [MTH89, Pau91]. It is *polymorphic*, so functions such as mean and variance can be coded to work over values of any dimension. The polymorphism is *implicit*—dimension variables are implicitly quantified in the same way as ML type variables. It is possible for the system to *infer* dimension types automatically, as well as check types which the programmer specifies.

Although it is described as an extension to ML, any language with an ML-like type system would suffice; indeed, it could even be added as an extension to a monomorphically-typed language as House did with Pascal. It is a conservative extension to ML in the sense that ML-typable programs remain typable, though functions may be given a more refined type than before.

We start with a set of *base* dimensions such as mass, length, and time, perhaps represented by the identifiers M, L and T as is conventional. Dimensions are written inside square brackets, for example $[MLT^{-2}]$. This notation cannot be confused with the ML list value shorthand, although some languages such as Haskell use $[\tau]$ to denote the list *type*.

For polymorphic dimensions we need dimension variables. We use d_1, d_2, \dots to distinguish them from ordinary type variables α, β, \dots . The unit dimension (for dimensionless quantities) is indicated by $[1]$.

We assume some kind of construct for declaring base dimensions. This could be extended to provide derived dimensions; we do not discuss this possibility here. The provision of multiple units for a single dimension is also an easy extension to the system.

3.1 Dimension types

We introduce new numeric types parameterised on dimension. The most obvious candidates are `real` and `complex`, with speeds having type $[LT^{-1}] \text{ real}$ and electric current [Current] `complex`. The parameter is written to the left of the type constructor in the style of Standard ML.

For the remainder of this paper we will only consider a single type constructor. In a type of the form $[\delta] \text{ real}$, δ is a dimension expression which is completely separate from other type-forming expressions and which may only appear as a parameter to numeric types.

3.2 Arithmetic

We give the following type schemes to the standard arithmetic operations:

<code>+, -</code>	$: \forall d. [d] \text{ real} \times [d] \text{ real} \rightarrow [d] \text{ real}$
<code>*</code>	$: \forall d_1 d_2. [d_1] \text{ real} \times [d_2] \text{ real} \rightarrow [d_1 d_2] \text{ real}$
<code>/</code>	$: \forall d_1 d_2. [d_1] \text{ real} \times [d_2] \text{ real} \rightarrow [d_1 d_2^{-1}] \text{ real}$
<code>sqrt</code>	$: \forall d. [d^2] \text{ real} \rightarrow [d] \text{ real}$
<code>exp, ln, sin, cos, tan</code>	$: [1] \text{ real} \rightarrow [1] \text{ real}$

It is often useful to coerce an integer into a dimensionless real, for which we provide a suitable function:

$$\text{real} : \text{int} \rightarrow [1] \text{ real}$$

Finally, it turns out that we need a polymorphic zero:

$$\text{zero} : \forall d. [d] \text{ real}$$

3.3 Some examples

Use of zero. Without a polymorphic zero value we would not even be able to test the sign of a number, for example, in an absolute value function:

```
fun abs x = if x < zero then zero-x else x
```

with type $\forall d. [d] \text{ real} \rightarrow [d] \text{ real}$. It is also essential as an identity for addition in functions such as the following:

```
fun sum []      = zero
  | sum (x::xs) = x + sum xs;
```

This has the type scheme $\forall d. [d] \text{ real list} \rightarrow [d] \text{ real list}$.

Statistical functions. Statistics provides a nice set of example functions because we would want to apply them over a large variety of differently dimensioned quantities. We list the code for mean and variance functions:

```
fun mean xs = sum xs / real (length xs);
fun variance xs =
  let val n = real (length xs)
      val m = mean xs
  in sum (map (fn x => sqr (x - m)) xs) / (n - real 1) end;
```

Their principal types, with those of some other statistical functions, are:

mean	: $\forall d. [d] \text{real list} \rightarrow [d] \text{real}$
variance	: $\forall d. [d] \text{real list} \rightarrow [d^2] \text{real}$
sdeviation	: $\forall d. [d] \text{real list} \rightarrow [d] \text{real}$
skewness	: $\forall d. [d] \text{real list} \rightarrow [1] \text{real}$
correlation	: $\forall d_1 d_2. [d_1] \text{real list} \rightarrow [d_2] \text{real list} \rightarrow [1] \text{real}$

Differentiation. We can write a function which differentiates another function numerically. It accepts a function f as argument and returns a new function which is the differential of f . We must also provide an increment h .

```
fun diff h f = fn x => (f (x+h) - f (x-h)) / (real 2 * h)
```

This has type scheme

$$\forall d_1 d_2. [d_1] \text{real} \rightarrow ([d_1] \text{real} \rightarrow [d_2] \text{real}) \rightarrow ([d_1] \text{real} \rightarrow [d_2 d_1^{-1}] \text{real})$$

Unlike the statistical examples, the type of the result is related to the type of more than one argument.

Root finding. Here is a tiny implementation of the Newton-Raphson method for finding roots of equations:

```
fun newton (f, f', x, eps) =
  let val dx = f x / f' x
      val x' = x - dx
  in if abs dx < eps then x' else newton (f, f', x', eps) end;
```

It accepts a function f , its derivative f' , an initial guess x and an accuracy eps . Its type is

$$\forall d_1 d_2. ([d_1] \text{real} \rightarrow [d_2] \text{real}) \times ([d_1] \text{real} \rightarrow [d_1^{-1} d_2] \text{real}) \times [d_1] \text{real} \times [d_1] \text{real} \rightarrow [d_1] \text{real}$$

Powers. To illustrate a more unusual type, here is a function of three arguments.

```
fun f (x,y,z) = x*x + y*y*y + z*z*z*z*z
```

This has the inferred type scheme

$$\forall d. [d^{15}] \text{real} \times [d^{10}] \text{real} \times [d^6] \text{real} \rightarrow [d^{30}] \text{real}$$

4 A dimension type system

We formalise the system by considering a very small ML-like language. Dimension expressions are defined by:

$$\delta ::= d \mid B \mid \delta \cdot \delta \mid \delta^{-1} \mid \mathbf{1}$$

where B is any base dimension and d is any dimension variable. The shorthand d^n ($n \in \mathbb{N}$) will be used to stand for the n -fold product of d with itself, and occasionally we will write $d_1 d_2$ instead of $d_1 \cdot d_2$.

Now we define *monomorphic* type expressions by:

$$\tau ::= \alpha \mid [\delta] \mathbf{real} \mid \tau \rightarrow \tau$$

where α is any type variable. *Polymorphic* type expressions, also called *type schemes* are defined by

$$\sigma ::= \tau \mid \forall \alpha. \sigma \mid \forall d. \sigma$$

We have extended the usual ML-style type schemes with quantification over dimension variables, which must be distinct from type variables in order to distinguish the two kinds of quantification. The flavour of polymorphism used for dimension types is the same as ordinary ML-like polymorphism. This leads to the usual problems but does mean that inference is straightforward. We shall have more to say on this subject later.

Finally, expressions are defined by

$$e ::= x \mid n \mid ee \mid \lambda x. e \mid \text{let } x = e \text{ in } e$$

where x is a variable and n is a real-valued constant such as 3.14. The full set of inference rules is now given, based on Cardelli [Car87]. Only two new rules are required—generalisation and specialisation for dimension quantification. A_x denotes the type assignment obtained from A by removing any typing statement for x .

VAR	$\frac{}{A \vdash x : \sigma} A(x) = \sigma$	REAL	$\frac{}{A \vdash n : [\mathbf{1}] \mathbf{real}} A \vdash n : \mathbf{real}$
GEN	$\frac{A \vdash e : \sigma}{A \vdash e : \forall \alpha. \sigma} \quad \alpha \text{ not free in } A$	SPEC	$\frac{A \vdash e : \forall \alpha. \sigma}{A \vdash e : \sigma[\tau/\alpha]} \quad A \vdash e : \forall d. \sigma$
DGEN	$\frac{A \vdash e : \sigma}{A \vdash e : \forall d. \sigma} \quad d \text{ not free in } A$	DSPEC	$\frac{A \vdash e : \forall d. \sigma}{A \vdash e : \sigma[\delta/d]} \quad A \vdash e : \forall \alpha. \sigma$
ABS	$\frac{A_x \cup \{x : \tau\} \vdash e : \tau'}{A \vdash \lambda x. e : \tau \rightarrow \tau'}$	APP	$\frac{A \vdash e : \tau \rightarrow \tau' \quad A \vdash e' : \tau}{A \vdash ee' : \tau'}$
LET	$\frac{A \vdash e : \sigma \quad A_x \cup \{x : \sigma\} \vdash e' : \tau}{A \vdash \text{let } x = e \text{ in } e' : \tau}$		

In addition to these rules we have equations relating dimensions:

$$\begin{aligned}\delta_1\delta_2 &=D \delta_2\delta_1 && (\text{commutativity}) \\ (\delta_1\delta_2)\delta_3 &=D \delta_1(\delta_2\delta_3) && (\text{associativity}) \\ \mathbf{1} \cdot \delta &=D \delta && (\text{identity}) \\ \delta\delta^{-1} &=D \mathbf{1} && (\text{inverses})\end{aligned}$$

and an inference rule relating equivalent types:

$$\text{DEQ} \frac{A \vdash e : \tau_1 \quad \vdash \tau_1 =_D \tau_2}{A \vdash e : \tau_2}$$

where $=_D$ is lifted to types by the obvious congruence.

It will be observed that none of the rules explicitly introduces types involving base dimensions. We assume that there is a means of declaring constants which represent a base unit for a particular base dimension. For the length dimension, for example, we might have a constant `metre` of type [L] `real`.

5 Dimensional Type Inference

5.1 Unification—algorithm *Unify*

At the heart of most type inference algorithms is the process of *unification*. Given an equation of the form

$$\tau_1 \stackrel{?}{=} \tau_2$$

we wish to find the *most general unifier*, a substitution S such that

1. $S(\tau_1) = S(\tau_2)$
2. For any other unifier S' there is a substitution S'' such that $S'' \circ S = S'$.

If equality is purely syntactic, there is a straightforward algorithm first devised by Robinson. It accepts a pair of types τ_1 and τ_2 and returns their most general unifier or fails if there is none.

$$\text{Unify}(\alpha, \alpha) \quad = \text{the identity substitution}$$

$$\text{Unify}(\alpha, \tau) = \text{Unify}(\tau, \alpha) = \begin{cases} \text{if } \alpha \text{ is in } \tau \text{ then fail (no unifier exists)} \\ \text{else return the substitution } \{\alpha \mapsto \tau\} \end{cases}$$

$$\begin{aligned}\text{Unify}(\tau_1 \rightarrow \tau_2, \tau_3 \rightarrow \tau_4) &= S_2 \circ S_1 \\ &\text{where } S_1 = \text{Unify}(\tau_1, \tau_3) \\ &\text{and } S_2 = \text{Unify}(S_1(\tau_2), S_1(\tau_4))\end{aligned}$$

To extend this to deal with types of the form $[\delta] \text{ real}$, we unify dimensions using another algorithm *DimUnify*. The additional clause is simply

$$\text{Unify}([\delta_1] \text{ real}, [\delta_2] \text{ real}) = \text{DimUnify}(\delta_1, \delta_2)$$

5.2 Dimensional Unification—algorithm *DimUnify*

We require an algorithm *DimUnify* which accepts two dimension expressions δ_1 and δ_2 and returns a substitution S over the dimension variables in the expressions such that

1. $S(\delta_1) =_D S(\delta_2)$
2. For any other unifier S' there is a substitution S'' such that $S'' \circ S =_D S'$.

This kind of unification is sometimes called *equational*, in contrast to ordinary Robinson unification which is *syntactic* or *free*. In our dimension type system, we want to unify with respect to the four laws listed earlier: associativity, commutativity, identity and inverses. It turns out that this particular brand of unification is decidable and *unitary* [Baa89, Nut90]: there is a single most general unifier if one exists at all. This has the consequence that, as for ML polymorphic types, if an expression is typable then it has a most general type from which any other type may be derived by simple substitution for dimension variables.

We will use Lankford's algorithm for Abelian group unification [LBB84]. It relies on the solution of linear equations in integers, for which there exist several algorithms including one by Knuth [Knu69]. Our treatment is slightly different in that we consider only a single equation.

First we transform the equation to the normalised form

$$d_1^{x_1} \cdot d_2^{x_2} \cdots d_m^{x_m} \cdot B_1^{y_1} \cdot B_2^{y_2} \cdots B_n^{y_n} \stackrel{?}{=} D \mathbf{1}$$

where d_i and B_j are distinct dimension variables and base dimensions.

Start by setting S to the empty substitution. If $m = 0$ and $n = 0$ then we are finished already. If $m = 0$ and $n \neq 0$ then fail: there is no unifier. Otherwise, find the dimension variable with exponent x_k of smallest absolute value in the equation. If x_k is negative, first take reciprocals of both sides by negating all exponents. Without loss of generality, we can assume that $k = 1$.

1. If $\forall i. x_i \bmod x_1 = 0$ and $\forall j. y_j \bmod x_1 = 0$, then the unifier is the following, composed with S .

$$d_1 \mapsto d_2^{-x_2/x_1} \cdots d_m^{-x_m/x_1} \cdot B_1^{-y_1/x_1} \cdots B_n^{-y_n/x_1}$$

2. Otherwise introduce a new variable d and compose with S the substitution

$$d_1 \mapsto d \cdot d_2^{-\lfloor x_2/x_1 \rfloor} \cdots d_m^{-\lfloor x_m/x_1 \rfloor} \cdot B_1^{-\lfloor y_1/x_1 \rfloor} \cdots B_n^{-\lfloor y_n/x_1 \rfloor}$$

to transform the equation to

$$d^{x_1} \cdot d_2^{x_2 \bmod x_1} \cdots d_m^{x_m \bmod x_1} \cdot B_1^{y_1 \bmod x_1} \cdots B_n^{y_n \bmod x_1} \stackrel{?}{=} D \mathbf{1}$$

If at this stage there are no variables in the equation other than d then there is no solution—no unifier exists.

Otherwise find the smallest exponent again and repeat the procedure.

This method must terminate because on each iteration we reduce the size of the smallest nonzero coefficient in the equation.

5.3 Inference—algorithm *Infer*

The type inference algorithm for ML is well-known and has been presented in many places. Our version differs in two respects—quantified dimension variables are instantiated at the same time as quantified type variables (when e is a variable), and generalization over free dimension variables is added to the usual generalization over free type variables (when e is a let-expression).

Given a type assignment A and an expression e , the algorithm *Infer* determines a pair (S, τ) where τ is the most general type of e and S is a substitution over the type and dimension variables in A under which this is true.

$$\text{Infer}(A, x) = (I, \tau[d'_1/d_1, \dots, d'_m/d_m, \alpha'_1/\alpha_1, \dots, \alpha'_n/\alpha_n])$$

where

$A(x)$ is $\forall d_1 \dots d_m. \forall \alpha_1 \dots \alpha_n. \tau$

d'_1, \dots, d'_m are fresh dimension variables

$\alpha'_1, \dots, \alpha'_n$ are fresh type variables

$$\text{Infer}(A, e_1 e_2) = (S_3 \circ S_2 \circ S_1, S_3(\alpha))$$

where

$(S_1, \tau_1) = \text{Infer}(A, e_1)$

$(S_2, \tau_2) = \text{Infer}(S_1(A), e_2)$

$S_3 = \text{Unify}(S_2(\tau_1), \tau_2 \rightarrow \alpha)$

α is a fresh type variable

$$\text{Infer}(A, \lambda x. e) = (S, S(\alpha) \rightarrow \tau)$$

where

$(S, \tau) = \text{Infer}(A_x \cup \{x : \alpha\}, e)$

α is a fresh type variable

$$\text{Infer}(A, \text{let } x = e \text{ in } e') = (S_2 \circ S_1, \tau_2)$$

where

$(S_1, \tau_1) = \text{Infer}(A, e)$

$(S_2, \tau_2) = \text{Infer}(S_1(A_x) \cup \{x : \forall d_1, \dots, d_m. \forall \alpha_1, \dots, \alpha_n. \tau_1\}, e')$

d_1, \dots, d_m are free dimension variables in τ_1 not in $S_1(A)$

$\alpha_1, \dots, \alpha_n$ are free type variables in τ_1 not in $S_1(A)$

The algorithm's correctness is shown by two theorems [Lei83, Dam85].

Theorem 1 (Soundness of Infer). *If $\text{Infer}(A, e)$ succeeds with result (S, τ) then there is a derivation of $S(A) \vdash e : \tau$.*

Theorem 2 (Syntactic Completeness of Infer). *If there is a derivation of $S(A) \vdash e : \tau$ then $\text{Infer}(A, e)$ is a principal typing for e , i.e. it succeeds with result (S_0, τ_0) and $S =_D S' \circ S_0$, $\tau =_D S'(\tau_0)$ for some substitution S' .*

To prove these theorems we first devise a syntax-oriented version of the inference rules and prove that they are equivalent to the rules given here. Then the proofs follow more straightforwardly by induction on the structure of e ; these will appear in a fuller version of this paper.

6 Implementation

The dimension type system described in this article has been implemented as an extension to the ML Kit compiler [Rot92], which is a full implementation of Standard ML as defined in [MTH89].

In order to fit naturally with the rest of Standard ML, the concrete syntax of dimension types is necessarily messy. Dimension variables are distinguished from ordinary type variables and identifiers by an initial underline character, as in `_a`. Base dimensions are ordinary identifiers declared by a special construct. This might also be used to introduce constants representing the base units for the dimension specified, as mentioned in section 4:

```
dimension M unit kg;
dimension L unit metre;
dimension T unit sec;
```

It would be easy to extend this to permit derived dimensions, in a fashion similar to ML type definition.

Dimension expressions are enclosed in square brackets, as is conventional. This happens to fit nicely with the notation for parameterised types. The unit dimension is simply `[]`. Exponents are written after a colon (e.g. area is `[L:2]`) and product is indicated by simple concatenation (e.g. density is `[M L:~3]`).

Any new type or datatype may be parameterised by dimension, by type, or by a mixture of both. Assuming a built-in `real` type we could define `complex` by

```
datatype [_a] complex = make_complex of [_a] real * [_a] real
```

Built-in functions as defined in the prelude are given new types, for example:

```
val sqrt : [_a:2] real -> [_a] real
val sin : [] real -> [] real
val + : [_a] real * [_a] real -> [_a] real
val * : [_a] real * [_b] real -> [_a _b] real
```

The one major problem is ML's overloading of such functions. The Definition of Standard ML gives types such as `num*num -> num` to arithmetic and comparison functions. A type-checker must use the surrounding context to determine whether `num` is replaced by `real` or `int`. We want to give dimensionally polymorphic types to these functions. This makes the Definition's scheme unworkable, especially in the case of multiplication. The current implementation has alternative names for dimensioned versions of these operations.

7 Some Problems

7.1 Equivalent types

ML type inference determines a most general type, if there is one, up to renaming of type variables. For example, the type scheme $\forall\alpha\beta.\alpha \times \beta$ is equivalent to $\forall\alpha\beta.\beta \times \alpha$. This equivalence is easy for the programmer to understand.

For dimension types, we have principal types with respect to the equivalence relation $=_D$, but there is no obvious way of choosing a canonical representative for a given equivalence class—there is no “principal syntax”. Type scheme $\forall d_1 \dots d_n. \tau_1$ is equivalent to $\forall d_1 \dots d_n. \tau_2$ if there are substitutions S_1 and S_2 over the bound variables d_1 to d_n such that

$$\begin{aligned} S_1(\tau_1) &=_{\mathcal{D}} \tau_2 \\ \text{and} \\ S_2(\tau_2) &=_{\mathcal{D}} \tau_1 \end{aligned}$$

This is *not* just $=_D$ plus renaming of type and dimension variables. For example, the current implementation of the system described in this article assigns the following type scheme to the `correlation` example of section 3.3.

$$\forall d_1 d_2. [d_1] \text{ real list} \rightarrow [d_2 d_1^{-1}] \text{ real list} \rightarrow [1] \text{ real}$$

which is equivalent to

$$\forall d_1 d_2. [d_1] \text{ real list} \rightarrow [d_2] \text{ real list} \rightarrow [1] \text{ real}$$

by the substitutions $d_2 \mapsto d_2 d_1$ (forwards) and $d_2 \mapsto d_2 d_1^{-1}$ (backwards). The second of these types is obviously more “natural” but I do not know how to formalise this notion and modify the inference algorithm accordingly.

In some cases there does not even appear to be a most natural form for the type. The following expressions are different representations of the principal type scheme for the differentiation function of section 3.3.

$$\begin{aligned} \forall d_1 d_2. [d_1] \text{ real} &\rightarrow ([d_1] \text{ real} \rightarrow [d_2] \text{ real}) \rightarrow ([d_1] \text{ real} \rightarrow [d_2 d_1^{-1}] \text{ real}) \\ &\quad \text{and} \\ \forall d_1 d_2. [d_1] \text{ real} &\rightarrow ([d_1] \text{ real} \rightarrow [d_1 d_2] \text{ real}) \rightarrow ([d_1] \text{ real} \rightarrow [d_2] \text{ real}) \end{aligned}$$

7.2 Dependent types

Consider a function for raising real numbers to integral powers:

```
fun power 0 x = 1.0
| power n x = x * power (n-1) x
```

Because the dimension of the result depends on an integer *value*, our system cannot give any better type than the dimensionless

$$\text{int} \rightarrow [1] \text{ real} \rightarrow [1] \text{ real}$$

This seems rather limited, but variable exponents are in fact rarely seen in scientific programs except in dimensionless expressions such as power series. A *dependent* type system would give a more informative type to this function:

$$\forall d. \prod n \in \text{int}. [d] \text{ real} \rightarrow [d^n] \text{ real}$$

There are also functions which intuitively should have a static type expressible in this system, but which cannot be inferred. Geometric mean is one example.

It seems as though its type should be $\forall d. [d] \text{ real list} \rightarrow [d] \text{ real}$, like the arithmetic mean mentioned earlier. Unfortunately its definition makes use of `rpower` and `prod` both of which have dimensionless type:

```
fun rpower (x,y) = exp(y*ln x);
fun prod []      = 1.0
  | prod (x::xs) = x*prod xs;
fun gmean xs = rpower(prod xs, 1.0 / real (length xs))
```

7.3 Polymorphism

Recursive definitions in ML are not polymorphic: occurrences of a recursively defined function *inside* the body of its definition can only be used monomorphically. For the typical ML programmer this problem rarely manifests itself. Unfortunately it is a more serious irritation in our dimension type system.

```
fun prodlists ([] , [])      = []
  | prodlists (x::xs, y::ys) = (x*y) :: prodlists (ys,xs)
```

The function `prodlists` calculates products of corresponding elements in a pair of lists, but bizarrely switches the arguments on the recursive call. Naturally this makes no difference to the result, given the commutativity of multiplication, but whilst a version without the exchange is given a type scheme

$$\forall d_1 d_2. [d_1] \text{ real list} \times [d_2] \text{ real list} \rightarrow [d_1 d_2] \text{ real list}$$

the version above has the less general

$$\forall d. [d] \text{ real list} \times [d] \text{ real list} \rightarrow [d^2] \text{ real list}$$

An analogous example in Standard ML is the (useless) function shown here:

```
fun funny c x y = if c=0 then 0 else funny (c-1) y x
```

This has inferred type $\forall \alpha. \text{int} \rightarrow \alpha \rightarrow \alpha \rightarrow \text{int}$ but might be expected to have the more general type $\forall \alpha \beta. \text{int} \rightarrow \alpha \rightarrow \beta \rightarrow \text{int}$. Extensions to the ML type system to permit polymorphic recursion have been proposed. It has been shown that the inference problem for such a system is undecidable [Hen93, KTU93].

The lack of polymorphic lambda-abstraction also reduces the generality of inferred types:

```
fun twice f x = f (f x);
fun sqr x      = x*x;
fun fourth x   = (twice sqr) x;
```

The following type schemes are assigned:

<code>twice</code>	$: \forall \alpha. (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$
<code>sqr</code>	$: \forall d. [d] \text{ real} \rightarrow [d^2] \text{ real}$
<code>fourth</code>	$: [1] \text{ real} \rightarrow [1] \text{ real}$

We would like `fourth` to have type $\forall d. [d] \text{ real} \rightarrow [d^4] \text{ real}$ but cannot have it because this would require `sqr` to be used at two different instances inside `twice`, namely $\forall d. [d] \text{ real} \rightarrow [d^2] \text{ real}$ and $\forall d. [d^2] \text{ real} \rightarrow [d^4] \text{ real}$.

This is a serious problem but not unpredictable so long as the programmer fully understands the nature of ML-style polymorphism. The same situation occurs in ordinary ML if we change the definition of `sqr` to be (x, x) . This time we expect `fourth` to have the type $\forall \alpha. \alpha \rightarrow (\alpha \times \alpha) \times (\alpha \times \alpha)$ but the expression is untypable because `sqr` must be used at the two instances $\forall \alpha. \alpha \rightarrow \alpha \times \alpha$ and $\forall \alpha. \alpha \times \alpha \rightarrow (\alpha \times \alpha) \times (\alpha \times \alpha)$. In fact, we cannot even write such a term in the second-order lambda calculus. It requires either a higher-order type system such as F_ω , or a system with intersection types, in which we could give `twice` the type

$$\forall \alpha \beta \gamma. (\alpha \rightarrow \beta) \wedge (\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \gamma)$$

and pass in `sqr` at two instances.

8 Related work

8.1 House's extension to Pascal

Before Wand and O'Keefe's recent work, the only attempt at a polymorphic dimension type system was the extension to Pascal proposed by House [Hou83]. In that system, types in procedure declarations may include a kind of dimension variable, as in the following example:

```
function ratio(a : real newdim u; b : real newdim v)
  : real dim u/v;
begin
  ratio := a/b
end;
```

Compared with modern notions of polymorphism, this is rather strange; the `newdim` construct introduces a new variable standing for some dimension, and `dim` makes use of already-introduced variables. It is as though `newdim` contains an implicit quantifier.

8.2 Wand and O'Keefe's system

Wand and O'Keefe define an ML-like type system extended with a single numeric type parameterised on dimension [WO91]. This takes the form $Q(n_1, \dots, n_N)$ where n_i are *number expressions* formed from number variables, rational constants, addition and subtraction operations, and multiplication by rational constants. It differs from the $[\delta] \text{ real}$ type of this paper in two ways:

1. A fixed number of base dimensions N is assumed. Dimension types are expressed as a N -tuple of number expressions, so if we have three base dimensions M , L and T , then $Q(n_1, n_2, n_3)$ represents the dimension $[M^{n_1} L^{n_2} T^{n_3}]$.

2. Dimensions have rational exponents. This means, for instance, that the type of the square root function can be expressed as

$$\forall i, j, k. Q(i, j, k) \rightarrow Q(0.5 * i, 0.5 * j, 0.5 * k)$$

in contrast to

$$\forall d. [d^2] \text{ real} \rightarrow [d] \text{ real}$$

in our system, and this function may be applied to a value of type $Q(1, 0, 0)$, whereas our system disallows its application to $[M] \text{ real}$.

Their inference algorithm, like ours, generates equations between dimensions. But in their system there are no “dimension constants” (our base dimensions) and equations are not necessarily integral, so Gaussian elimination is used to solve them.

Wand and O’Keefe’s types are unnecessarily expressive and can be nonsensical dimensionally. Consider the type $\forall i, j, k. Q(i, j, k) \rightarrow Q(i, 2 * j, k)$ which squares the length dimension but leaves the others alone, or $\forall i, j, k. Q(i, j, k) \rightarrow Q(j, i, k)$ which swaps the mass and length dimensions. Fortunately no expression in the language will be assigned such types. Also, non-integer exponents should not be necessary—polymorphic types can be expressed without them and values with fractional dimension exponents do not seem to occur in science.

They propose a construct `newdim` which introduces a *local* dimension. In our system the `dimension` declaration could perhaps be used in a local context, in the same way that the `datatype` construct of ML is used already.

The problem of finding canonical expressions for types presumably occurs in their system too, as well as the limitations of implicit polymorphism described here.

9 Conclusion and Future Work

The system described in this paper provides a natural way of adding dimensions to a polymorphically-typed programming language. It has been implemented successfully, and it would be straightforward to add features such as derived dimensions, local dimensions, and multiple units of measure within a single dimension.

To overcome the problems discussed in section 7 it might be possible to make the system more polymorphic, but *only* over dimensions in order to retain decidability. An alternative which is being studied is the use of intersection types.

So far no formal semantics has been devised for the system. This would be used to prove a result analogous to the familiar “well-typed programs cannot go wrong” theorem for ML.

Acknowledgements

This work was supported financially by a SERC Studentship. I would like to thank Alan Mycroft, Francis Davey, Nick Benton and Ian Stark for discussions on the subject of this paper, and the anonymous referees for their comments.

References

- [Baa89] F. Baader. Unification in commutative theories. *Journal of Symbolic Computation*, 8:479–497, 1989.
- [Bal87] G. Baldwin. Implementation of physical units. *SIGPLAN Notices*, 22(8):45–50, August 1987.
- [Car87] L. Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8(2):147–172, 1987.
- [Dam85] L. Damas. *Type Assignment in Programming Languages*. PhD thesis, Department of Computer Science, University of Edinburgh, 1985.
- [DMM86] A. Dreiheller, M. Moerschbacher, and B. Mohr. PHYSCAL—programming Pascal with physical units. *SIGPLAN Notices*, 21(12):114–123, December 1986.
- [Geh85] N. H. Gehani. Ada’s derived types and units of measure. *Software—Practice and Experience*, 15(6):555–569, June 1985.
- [Hen93] F. Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, April 1993.
- [Hou83] R. T. House. A proposal for an extended form of type checking of expressions. *The Computer Journal*, 26(4):366–374, 1983.
- [KL78] M. Karr and D. B. Loveman III. Incorporation of units into programming languages. *Communications of the ACM*, 21(5):385–391, May 1978.
- [Knu69] D. Knuth. *The Art of Computer Programming, Vol. 2*, pages 303–304. Addison-Wesley, 1969.
- [KTU93] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, April 1993.
- [Lan51] H. L. Langhaar. *Dimensional Analysis and Theory of Models*. John Wiley and Sons, 1951.
- [LBB84] D. Lankford, G. Butler, and B. Brady. Abelian group unification algorithms for elementary terms. *Contemporary Mathematics*, 29:193–199, 1984.
- [Lei83] D. Leivant. Polymorphic type inference. In *ACM Symposium on Principles of Programming Languages*, 1983.
- [Män86] R. Männer. Strong typing and physical units. *SIGPLAN Notices*, 21(3):11–20, March 1986.
- [Man87] R. Mankin. letter. *SIGPLAN Notices*, 22(3):13, March 1987.
- [MTH89] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Mass., 1989.
- [Nut90] W. Nutt. Unification in monoidal theories. In *10th International Conference on Automated Deduction*, volume 449 of *Lecture Notes in Computer Science*, pages 618–632. Springer-Verlag, July 1990.
- [Pau91] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.
- [Rot92] N. Rothwell. Miscellaneous design issues in the ML Kit. Technical Report ECS-LFCS-92-237, Laboratory for Foundations of Computer Science, University of Edinburgh, 1992.
- [WO91] M. Wand and P. M. O’Keefe. Automatic dimensional inference. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*. MIT Press, 1991.

A Synergistic Analysis for Sharing and Groundness which Traces Linearity

Andy King

Department of Electronics and Computer Science, **
The University of Southampton, Southampton, S09 5NH, UK.

Abstract. Accurate variable sharing information is crucial both in the automatic parallelisation and in the optimisation of sequential logic programs. Analysis for possible variable sharing is thus an important topic in logic programming and many analyses have been proposed for inferring dependencies between the variables of a program, for instance, by combining domains and analyses. This paper develops the combined domain theme by explaining how term structure, and in particular linearity, can be represented in a sharing group format. This enables aliasing behaviour to be more precisely captured; groundness information to be more accurately propagated; and in addition, refines the tracking and application of linearity. In practical terms, this permits aliasing and groundness to be inferred to a higher degree of accuracy than in previous proposals and also can speed up the analysis itself. Correctness is formally proven.

1 Introduction

Abstract interpretation for possible sharing is an important topic of logic programming. Sharing (or aliasing) analysis conventionally infers which program variables are definitely grounded and which variables can never be bound to terms containing a common variable. Applications of sharing analysis are numerous and include: the sound removal of the occur-check [22]; optimisation of backtracking [3]; the specialisation of unification [24]; and the elimination of costly checks in independent and-parallelism [20, 14, 21]. Early proposals for sharing analysis include [25, 10, 19].

This paper is concerned with a semantic basis for sharing analysis, and in particular, the justification of a high precision abstract unification algorithm. Following the approach of abstract interpretation [8], the abstract unification algorithm (the abstract operation) essentially mimics unification (the concrete operation) by finitely representing substitutions (the concrete data) with sharing abstractions (the abstract data). The accuracy of the analysis depends, in part, on the substitution properties that the sharing abstractions capture. Sharing abstractions usually capture groundness and aliasing information, and indeed, accurate analyses are often good at groundness propagation [14, 21]. A knowledge of groundness can improve sharing and *vice versa*. A synergistic relationship also

** New address: The Computing Laboratory, The University of Kent, Canterbury, CT2 7LX, UK.

exists between sharing and type analysis. Type analysis deduces structural properties of aggregate data. By keeping track of type information, that is inferring structural properties of substitutions, it is possible to infer more accurate sharing information. Conversely, more accurate type information can be deduced if sharing is traced.

Type information is often applied by combining sharing and freeness analysis [20, 7, 23] or by tracing linearity [22, 5]. Freeness information differentiates between a free variable, a variable which is definitely not bound to non-variable term; and a non-free variable, a variable which is possibly bound to a non-variable term. Freeness information is useful in its own right, in fact it is essential in the detection of non-strict and-parallelism [13]. A more general notion than freeness is linearity [22, 5]. Linearity relates to the number of times a variable occurs in a term. A term is linear if it definitely does not contain multiple occurrences of a variable; otherwise it is non-linear. Without exploiting linearity (or freeness), analyses have to assume that aliasing is transitive [5]. The significance of linearity is that the unification of linear terms only yields restricted forms of aliasing. Thus, if terms can be inferred to be linear, worst case aliasing need not be assumed in an analysis.

Sharing analyses can be used in isolation, but an increasing trend is to combine domains and analyses to improve accuracy [6]. For example, the pair-sharing domain of Søndergaard [22, 5], tracks linearity but is not so precise at propagating groundness information. Conversely, sharing group domains [14, 21] accurately characterise groundness but do not exploit linearity. The rationale behind [6], therefore, is to run multiple analyses in lock step. At each step, the sharing information from different analyses is compared and used to improve the precision. For instance, the linearity of the Søndergaard domain [22, 5] can be used to prune out spurious aliasing in the sharing group analysis [14, 21]; and the groundness information of the Jacobs and Langen domain can be used to remove redundant aliasing in the Søndergaard analysis.

This paper develops the combined domain theme by explaining how the linearity of the the Søndergaard domain [22, 5] can be represented in the sharing group format of the Jacobs and Langen domain [14, 21]. This enables both aliasing behaviour to be precisely captured, and groundness information to be accurately propagated, in a single coherent domain and analysis. This is not an exercise in aesthetics but has a number of important and practical implications:

1. By embedding linearity into sharing groups, the classic notion of linearity [22, 5] can be refined. Specifically, if a variable is bound to a non-linear term, it is still possible to differentiate between which variables of the term occur multiply in the term and which variables occur singly in the term. Put another way, the abstraction proposed in this paper records why a variable binding is potentially non-linear, rather than merely indicating that it is possibly non-linear. Previously, the variable would simply be categorised as non-linear, and worst-case aliasing assumed. The refined notion of linearity permits more accurate aliasing information to be squeezed out of the analysis. This can, in turn, potentially identify more opportunities for parallelism and optimisation.

2. Tracking aliasing more accurately can also improve the efficiency of the analysis [6]. Possible aliases are recorded and manipulated in a data structure formed from sharing groups. As the set of possible aliases is inferred more accurately, so the set becomes smaller, and thus the number of sharing groups is reduced. The size of the data structures used in the analysis are therefore pruned, and consequently, analysis can proceed more quickly.
 Moreover, the sharing abstractions defined in this paper are described in terms of a single domain and manipulated by a single analysis. This is significant because, unlike the multiple analyses approach [6], it avoids the duplication of abstract interpretation machinery and therefore simplifies the analysis. In practical terms, this is likely to further speedup the analysis [12]. Furthermore, the closure under union operation implicit in the analyses of [14, 21] has exponential time- and space-complexity in the number of sharing groups. It is therefore important to limit its use. In this paper, an analog of closure under union operation is employed, but is only applied very conservatively to a restricted subset of the set of sharing groups. This is also likely to contribute to faster analysis.
3. Errors and omissions have been reported [4, 9] in some of the more recent proposals for improving sharing analysis with type information [20, 7, 23]. Although the problems relate to unusual or rare cases, and typically the analyses can be corrected, these highlight that analyses are often sophisticated, subtle and difficult to get right. Thus, formal proof of correctness is useful, indeed necessary, to instill confidence. For the analysis described in this paper, safety has been formally proved. In more pragmatic terms this means that the implementor can trust the results given by the analysis.

The exposition is structured as follows. Section 2 describes the notation and preliminary definitions which will be used throughout. Also, linearity is formally introduced and its significance for aliasing is explained. In section 3, the focus is on abstracting data. A novel abstraction for substitutions is proposed which elegantly and expressively captures both linear and sharing properties of substitutions. In section 4, the emphasis changes to abstracting operations. Abstract analogs for renaming, unification, composition and restriction are defined in terms of an abstract *unify* operator [14]. An abstract unification algorithm is precisely and succinctly defined which, in turn, describes an abstract analog of *unify*. (Once an abstract *unify* operator is specified and proved safe, a complete and correct abstract interpreter is practically defined by virtue of existing abstract interpretation frameworks [1, 17, 21].) Finally, sections 5 and 6 present the related work and the concluding discussion. For reasons of brevity and continuity, proofs are not included in the paper, but can be found in [15].

2 Notation and preliminaries

To introduce the analysis some notation and preliminary definitions are required. The reader is assumed to be familiar with the standard constructs used in logic

programming [18] such as a universe of all variables ($u, v \in Uvar$); the set of terms ($t \in Term$) formed from the set of functors ($f, g, h \in Func$) (of the first-order language underlying the program); and the set of program atoms $Atom$. It is convenient to denote $f(t_1, \dots, t_n)$ by τ_n and $f'(t'_1, \dots, t'_n)$ by τ'_n . Also let $\tau_0 = f$ and $\tau'_0 = f'$. Let $Pvar$ denote a finite set of program variables - the variables that are in the text of the program; and let $var(o)$ denote the set of variables in a syntactic object o .

2.1 Substitutions

A substitution ϕ is a total mapping $\phi : Uvar \rightarrow Term$ such that its domain $dom(\phi) = \{u \in Uvar \mid \phi(u) \neq u\}$ is finite. The application of a substitution ϕ to a variable u is denoted by $\phi(u)$. Thus the codomain is given by $cod(\phi) = \bigcup_{u \in dom(\phi)} var(\phi(u))$. A substitution ϕ is sometimes represented as a finite set of variable and term pairs $\{u \mapsto \phi(u) \mid u \in dom(\phi)\}$. The identity mapping on $Uvar$ is called the empty substitution and is denoted by ϵ . Substitutions, sets of substitutions, and the set of substitutions are denoted by lower-case Greek letters, upper-case Greek letters, and $Subst$.

Substitutions are extended in the usual way from variables to functions, from functions to terms, and from terms to atoms. The restriction of a substitution ϕ to a set of variables $U \subseteq Uvar$ and the composition of two substitutions ϕ and φ , are denoted by $\phi \upharpoonright U$ and $\phi \circ \varphi$ respectively, and defined so that $(\phi \circ \varphi)(u) = \phi(\varphi(u))$. The preorder $Subst \sqsubseteq$, ϕ is more general than φ , is defined by: $\phi \sqsubseteq \varphi$ if and only if there exists a substitution $\psi \in Subst$ such that $\varphi = \psi \circ \phi$. The preorder induces an equivalence relation \approx on $Subst$, that is: $\phi \approx \varphi$ if and only if $\phi \sqsubseteq \varphi$ and $\varphi \sqsubseteq \phi$. The equivalence relation \approx identifies substitutions with consistently renamed codomain variables which, in turn, factors $Subst$ to give the poset $Subst/\approx \sqsubseteq$ defined by: $[\phi]_\approx \sqsubseteq [\varphi]_\approx$ if and only if $\phi \sqsubseteq \varphi$.

2.2 Equations and most general unifiers

An equation is an equality constraint of the form $a = b$ where a and b are terms or atoms. Let $(e \in) Eqn$ denote the set of finite sets of equations. The equation set $\{e\} \cup E$, following [5], is abbreviated by $e : E$. The set of most general unifiers of E , $mgu(E)$, is defined operationally [14] in terms of a predicate mgu . The predicate $mgu(E, \phi)$ which is true if ϕ is a most general unifier of E .

Definition 1 mgu . The set of most general unifiers $mgu(E) \in \wp(Subst)$ is defined by: $mgu(E) = \{\phi \mid mgu(E, \phi)\}$ where

$$\begin{aligned}
 & mgu(\emptyset, \epsilon) \\
 & mgu(v = v' : E, \zeta) \text{ if } mgu(E, \zeta) \wedge v \equiv v' \\
 & mgu(v = v' : E, \zeta \circ \eta) \text{ if } mgu(\eta(E), \zeta) \wedge v \not\equiv v' \wedge \eta = \{v \mapsto v'\} \\
 & mgu(v = v' : E, \zeta \circ \eta) \text{ if } mgu(\eta(E), \zeta) \wedge v \not\equiv v' \wedge \eta = \{v' \mapsto v\} \\
 & mgu(v = \tau_n : E, \zeta \circ \eta) \text{ if } mgu(\eta(E), \zeta) \wedge v \notin var(\tau_n) \wedge \eta = \{v \mapsto \tau_n\} \\
 & mgu(\tau_n = v : E, \zeta \circ \eta) \text{ if } mgu(\eta(E), \zeta) \wedge v \notin var(\tau_n) \wedge \eta = \{v \mapsto \tau_n\} \\
 & mgu(\tau_n = \tau'_n : E, \zeta) \text{ if } mgu(t_1 = t'_1 : \dots : t_n = t'_n : E, \zeta) \wedge f \equiv f'
 \end{aligned}$$

By induction it follows that $\text{dom}(\phi) \cap \text{cod}(\phi) = \emptyset$ if $\phi \in \text{mgu}(E)$, or put another way, that the most general unifiers are idempotent [16].

Following [14], the semantics of a logic program is formulated in terms of a single *unify* operator. To construct *unify*, and specifically to rename apart program variables, an invertible substitution [16], Υ , is introduced. It is convenient to let $Rvar \subseteq Uvar$ denote a set of renaming variables that cannot occur in programs, that is $Pvar \cap Rvar = \emptyset$, and suppose that $\Upsilon : Pvar \rightarrow Rvar$.

Definition 2 *unify*. The partial mapping $\text{unify} : Atom \times Subst/\approx \times Atom \times Subst/\approx \rightarrow Subst/\approx$ is defined by:

$$\text{unify}(a, [\phi]_\approx, b, [\psi]_\approx) = [(\varphi \circ \phi) \upharpoonright Pvar]_\approx \text{ where } \varphi \in \text{mgu}(\{\phi(a) = \Upsilon(\psi(b))\})$$

To approximate the *unify* operation it is convenient to introduce a collecting semantics, concerned with sets of substitutions, to record the substitutions that occur at various program points. In the collecting semantics interpretation, *unify* is extended to *unify*^c, which manipulates (possibly infinite) sets of substitutions.

Definition 3 *unify*^c. The mapping $\text{unify}^c : Atom \times \wp(Subst/\approx) \times Atom \times \wp(Subst/\approx) \rightarrow \wp(Subst/\approx)$ is defined by:

$$\text{unify}^c(a, \Phi, b, \Psi) = \{[\theta]_\approx \mid [\phi]_\approx \in \Phi \wedge [\psi]_\approx \in \Psi \wedge [\theta]_\approx = \text{unify}(a, [\phi]_\approx, b, [\psi]_\approx)\}$$

2.3 Linearity and substitutions

To be more precise about linearity, it is necessary to introduce the variable multiplicity of a term t , denoted $\chi(t)$.

Definition 4 **variable multiplicity**, χ [5]. The variable multiplicity operator $\chi : Term \rightarrow \{0, 1, 2\}$ is defined by:

$$\chi(t) = \max(\{\chi_u(t) \mid u \in Uvar\}) \text{ where } \chi_u(t) = \begin{cases} 0 & \text{if } u \text{ does not occur in } t \\ 1 & \text{if } u \text{ occurs only once in } t \\ 2 & \text{if } u \text{ occurs many times in } t \end{cases}$$

If $\chi(t) = 0$, t is ground; if $\chi(t) = 1$, t is linear; and if $\chi(t) = 2$, t is non-linear. The significance of linearity is that the unification of linear terms only yields restricted forms of aliasing. Lemma 5 states some of the restrictions on a most general unifier which follow from unification with a linear term.

Lemma 5. $\chi(b) \neq 2 \wedge \text{var}(a) \cap \text{var}(b) = \emptyset \wedge \phi \in \text{mgu}(\{a = b\}) \Rightarrow$

1. $\forall u \in Uvar. \chi(\phi(u)) = 2 \Rightarrow u \in \text{var}(b)$
2. $\forall u, u' \in Uvar. u \neq u' \wedge \text{var}(\phi(u)) \cap \text{var}(\phi(u')) \neq \emptyset \Rightarrow u \notin \text{var}(a) \vee u' \notin \text{var}(a)$.
3. $\forall u', u'' \in \text{var}(b). u' \neq u'' \wedge w \in \text{var}(\phi(u')) \cap \text{var}(\phi(u'')) \Rightarrow \exists u \in \text{var}(a). \chi_u(a) = 2 \wedge w \in \text{var}(\phi(u))$

Application of lemma 5 is illustrated in example 1.

Example 1. Note that $\phi \in mgu(\{f(u, v, v) = f(x, y, z)\})$ where $\phi = \{v \mapsto y, x \mapsto u, z \mapsto y\}$, $\chi(f(x, y, z)) \neq 2$ and that $f(u, v, v)$ and $f(x, y, z)$ do not share variables. Observe that

1. The variables u and v of $f(u, v, v)$ remain linear after unification, that is, $\chi(\phi(u)) = 1$ and $\chi(\phi(v)) = 1$, as predicted by case 1 of lemma 5.
2. The variables of $f(u, v, v)$, specifically u and v , remain unaliased after unification. Indeed, case 2 of lemma 5 asserts that since $u, v \in var(f(u, v, v))$, $var(\phi(u)) \cap var(\phi(v)) = \emptyset$.
3. Informally, case 3 of lemma 5 states that the aliasing which occurs between the variables of $f(x, y, z)$, is induced by a variable of $f(u, v, v)$ which has a multiplicity of 2. For instance, $y \in var(\phi(y)) \cap var(\phi(z))$ with $\chi_v(f(u, v, v)) = 2$ and $y \in var(\phi(v))$.

Lemma 5 differs from the corresponding lemma in [5] (lemma 2.2) in two ways. First, lemma 5 requires that a and b do not share variables. This is essentially a work-around for a subtle mistake in lemma 2.2 [9]. Second, lemma 5 additionally states that a variable which only occurs once in a can only be aliased to one variable in b . This observation permits linearity to be exploited further than in the original proposals for tracking sharing with linearity [22, 5] by putting a tighter constraint of the form of aliasing that occurs on unification with a linear term. The proof for lemma 5 follows by induction on the steps of the unification algorithm.

3 Abstracting substitutions

Sharing analysis is primarily concerned with characterising the sharing effects that can arise among program variables. Correspondingly, abstract substitutions are formulated in terms of sharing groups [14] which represent which program variables share variables. Formally, an abstract substitution is structured as a set of sharing groups where a sharing group is a (possibly empty) set of program variable and linearity pairs.

Definition 6 Occ_{Svar} . The set of sharing groups, $(o \in) Occ_{Svar}$, is defined by:

$$Occ_{Svar} = \{o \in \wp(Svar \times \{1, 2\}) \mid \forall u \in Svar . \langle u, 1 \rangle \notin o \vee \langle u, 2 \rangle \notin o\}$$

$Svar$ is a finite set of program variables. The intuition is that a sharing group records which program variables are bound to terms that share a variable. Additionally, a sharing group expresses how many times the shared variable occurs in the terms to which the program variables are bound. Specifically, a program variable is paired with 1 if it is bound to a term in which the shared variable only occurs once. The variable is paired with 2 if it can be bound to a term in which the shared variable occurs possibly many times. The finiteness of Occ_{Svar} follows from the finiteness of $Svar$. ($Svar$ usually corresponds to $Pvar$, the set of

program variables. It is necessary to parameterise Occ , however, so that abstract substitutions are well-defined under renaming by Υ . Then $Svar = Rvar$.)

The precise notion of abstraction is first defined for a single substitution via lin and then, by lifting lin , generalised to sets of substitutions.

Definition 7 *occ* and *lin*. The abstraction mappings $occ : Uvar \times Subst \rightarrow Occ_{Svar}$ and $lin : Subst/\approx \rightarrow \wp(Occ_{Svar})$ are defined by:

$$occ(u, \phi) = \{\langle v, \chi_u(\phi(v)) \rangle \mid u \in var(\phi(v)) \wedge v \in Svar\}$$

$$lin([\phi]_\approx) = \{occ(u, \phi) \mid u \in Uvar\}$$

The mapping lin is well-defined since $lin([\phi]_\approx) = lin([\varphi]_\approx)$ if $\phi \approx \varphi$. The mapping occ is defined in terms of $Svar$ because, for the purposes of analysis, the only significant bindings are those which relate to the program variables (and renamed program variables). Note that $\emptyset \in lin([\phi]_\approx)$ since the codomain of a substitution is always finite.

The abstraction lin is analogous to the abstraction \mathcal{A} used in [21] and implicit in [14]. Both abstractions are formulated in terms of sharing groups. The crucial difference is that lin , as well as expressing sharing, additionally represents linearity information.

Example 2. Suppose $Svar = \{u, v, w, x, y, z\}$ and $\phi = \{u \mapsto u_1, w \mapsto v, x \mapsto f, y \mapsto g(u_1, u_2, u_2), z \mapsto h(u_2, u_3, u_3)\}$ then

$$\begin{aligned} lin([\phi]_\approx) &= \{\emptyset, occ(u_1, \phi), occ(u_2, \phi), occ(u_3, \phi), occ(v, \phi)\} = \\ &= \{\emptyset, \{\langle u, 1 \rangle, \langle y, 1 \rangle\}, \{\langle y, 2 \rangle, \langle z, 1 \rangle\}, \{\langle z, 2 \rangle\}, \{\langle v, 1 \rangle, \langle w, 1 \rangle\}\} \end{aligned}$$

since $occ(w, \phi) = occ(x, \phi) = occ(y, \phi) = occ(z, \phi) = \emptyset$. The salient properties of ϕ , namely sharing, groundness and linearity, are all captured by $lin([\phi]_\approx)$. The variables of $Svar$ which ϕ grounds, do not appear in $lin([\phi]_\approx)$; and the variables of $Svar$ which are independent (unaliased), never occur in the same sharing group of $lin([\phi]_\approx)$. Thus $lin([\phi]_\approx)$ indicates that x is ground and that, for example, v and y are independent. Additionally, $lin([\phi]_\approx)$ captures the fact that grounding either v or w grounds the other. Or, put another way, that v and w are strongly coupled [25].

Linearity is also represented and $lin([\phi]_\approx)$ indicates that $\chi(\phi(x)) = 0; \chi(\phi(u)) = \chi(\phi(v)) = \chi(\phi(w)) = 1$; and $\chi(\phi(y)) = \chi(\phi(z)) = 2$. It is evident that $\chi(\phi(w)) = 1$, for instance, since $\chi_v(\phi(w)) = 1$ and $\chi_u(\phi(w)) \neq 2$ for all $u \in Uvar$. Specifically, $\langle w, 1 \rangle \in occ(v, \phi)$ and $\langle w, 2 \rangle \notin occ(u, \phi)$ for all $u \in Uvar$. The subtlety is that the domain represents variable multiplicity information slightly more accurately than the Søndergaard domain [22, 5]. Note that although $\chi(\phi(y)) = 2$ and y is aliased to both u and z , $lin([\phi]_\approx)$ indicates that the variable that occurs through u and y (namely u_1) occurs only once in $\phi(y)$ whereas the variable through y and z (that is to say u_2) occurs multiply in $\phi(y)$. This can be exploited to gain more precise analysis.

The abstract domain, the set of abstract substitutions, is defined below using the convention that abstractions of concrete objects and operations are distinguished with a * from the corresponding concrete object or operation.

Definition 8 $\text{Subst}_{\text{Svar}}^*$. The set of abstract substitutions, $\text{Subst}_{\text{Svar}}^*$, is defined by: $\text{Subst}_{\text{Svar}}^* = \wp(\text{Occ}_{\text{Svar}})$.

Like previous sharing groups domains [14, 21], $\text{Subst}_{\text{Svar}}^*$ (\subseteq) is a finite lattice with set union as the lub. $\text{Subst}_{\text{Svar}}^*$ is finite since Occ_{Svar} is finite.

The *lin* abstraction naturally lifts to sets of substitutions, but to define concretisation, the notion of approximation implicit in linearity (specifically in the denotations 1 and 2) must be formalised. In the abstraction, a program variable is paired with 1 if it is definitely bound to a term in which the shared variable only occurs once; and is paired with 2 if it can possibly be bound to a term in which the shared variable occurs multiply. This induces the poset $\text{Occ}_{\text{Svar}}(\leq)$ defined by: $o \leq o'$ if and only if $\text{var}(o) = \text{var}(o')$ and for all $\langle u, m \rangle \in o$ there exists $\langle u, m' \rangle \in o'$ such that $m \leq m'$. The poset lifts to the preorder $\text{Subst}_{\text{Svar}}^*(\leq)$ by: $\phi^* \leq \phi'^*$ if and only if for all $o \in \phi^*$ there exists $o' \in \phi'^*$ such that $o \leq o'$.

Definition 9 α_{lin} and γ_{lin} . The abstraction and concretisation mappings $\alpha_{\text{lin}} : \wp(\text{Subst}/\approx) \rightarrow \text{Subst}_{\text{Svar}}^*$ and $\gamma_{\text{lin}} : \text{Subst}_{\text{Svar}}^* \rightarrow \wp(\text{Subst}/\approx)$ are defined by:

$$\alpha_{\text{lin}}(\Phi) = \bigcup_{[\phi]_{\approx} \in \Phi} \text{lin}([\phi]_{\approx}), \quad \gamma_{\text{lin}}(\phi^*) = \{[\phi]_{\approx} \in \text{Subst}/\approx \mid \text{lin}([\phi]_{\approx}) \leq \phi^*\}$$

The structure of α_{lin} and γ_{lin} mirrors that of the abstraction and concretisation operations found in [14, 21].

As illustrated in example 2, the *lin* abstraction can encode the variable multiplicity of a substitution. More significantly, if $\phi \in \gamma_{\text{lin}}(\phi^*)$, the variable multiplicity of $\phi(t)$ can be (partially) deduced from t and ϕ^* . The precise relationship between $\chi(\phi(t))$ and t and ϕ^* is formalised in definition 10 and lemma 11, with an analog of χ , denoted χ^* .

Definition 10 χ^* . The abstract variable multiplicity operator $\chi^* : \text{Term} \times \text{Occ}_{\text{Svar}} \rightarrow \{0, 1, 2\}$ is defined by:

$$\chi^*(t, o) = \begin{cases} 0 & \text{if } \forall v \in \text{var}(o) \cdot \chi_v(t) = 0 \\ 2 & \text{if } \exists v \in \text{var}(o) \cdot \chi_v(t) = 2 \\ 2 & \text{if } \exists v, v' \in \text{var}(t) \cdot v, v' \in \text{var}(o) \wedge v \neq v' \\ 2 & \text{if } \exists v \in \text{var}(t) \cdot (v, 2) \in o \\ 1 & \text{otherwise} \end{cases}$$

Lemma 11.

$$\text{var}(t) \subseteq \text{Svar} \wedge \text{occ}(u, \phi) \leq o \Rightarrow \chi_u(\phi(t)) \leq \chi^*(t, o)$$

To conservatively calculate the variable multiplicity of a term t in the context of a set of substitutions represented by ϕ^* , the sharing group operator χ^* is lifted to abstract substitutions via *ln* and *nl*.

Definition 12 *ln* and *nl*. The mappings $ln : Term \times Subst_{Svar}^* \rightarrow Subst_{Svar}^*$ and $nl : Term \times Subst_{Svar}^* \rightarrow Subst_{Svar}^*$ are defined by:

$$ln(t, \phi^*) = \{o \in \phi^* \mid \chi^*(t, o) = 1\}, \quad nl(t, \phi^*) = \{o \in \phi^* \mid \chi^*(t, o) = 2\}$$

The operators *ln* and *nl* essentially categorise ϕ^* into two sorts of sharing group: sharing groups which describe aliasing for which $\phi(t)$ is definitely linear; and sharing groups which represent aliasing for which $\phi(t)$ is possibly non-linear. An immediate corollary of lemma 11, corollary 13, asserts that $\phi(t)$ is linear if $nl(t, \phi^*)$ is empty.

Corollary 13.

$$[\phi]_{\approx} \in \gamma_{lin}(\phi^*) \wedge var(t) \subseteq Svar \wedge nl(t, \phi^*) = \emptyset \Rightarrow \chi(\phi(t)) \neq 2$$

The significance of corollary 13 is that it explains how by inspecting t and ϕ^* , $\phi(t)$ can be inferred to be linear, thereby enabling linear instances of unification to be recognised.

4 Abstracting unification

The collecting version of the *unify* operator, $unify^c$, provides a basis for abstracting the basic operations of logic programming by spelling out how to manipulate (possibly infinite) sets of substitutions. The usefulness of the collecting semantics as a form of program analysis, however, is negated by the fact that it can lead to non-terminating computations. Therefore, in order to define a practical analyser it is necessary to finitely abstract $unify^c$. To synthesise a sharing analysis, an analog of $unify^c$, $unify^*$, is introduced to manipulate sets of substitutions following the abstraction scheme prescribed by α_{lin} and γ_{lin} .

Just as $unify^c$ is defined in terms of *mgu*, $unify^*$ is defined in terms of an abstraction of *mgu*, *mge*, which traces the steps of the unification algorithm. The unification algorithm takes as input, E , a set of unification equations. E is recursively transformed to a set of simplified equations which assume the form $v = v'$ or $v = \tau_n$. These simplified equations are then solved. The equation solver *mge*, adopts a similar strategy, but relegates the solution of the simplified equations to *solve*. The skeleton of the abstract equation solver *mge* is given below in definition 14.

Definition 14 *mge*. The relation $mge : Eqn \times Subst_{Svar}^* \times Subst_{Svar}^* \rightarrow Subst_{Svar}^*$ is defined by:

$$\begin{aligned} mge(\emptyset, \sigma^*, \theta^*) \\ mge(v = v' : E, \sigma^*, \theta^*) \text{ if } mge(E, \sigma^*, \theta^*) \wedge & \quad v \equiv v' \\ mge(v = v' : E, \sigma^*, \theta^*) \text{ if } mge(E, solve(v, v', \sigma^*), \theta^*) \wedge & \quad v \not\equiv v' \\ mge(v = \tau_n : E, \sigma^*, \theta^*) \text{ if } mge(E, solve(v, \tau_n, \sigma^*), \theta^*) \wedge & \quad v \notin var(\tau_n) \\ mge(\tau_n = v : E, \sigma^*, \theta^*) \text{ if } mge(v = \tau_n : E, \sigma^*, \theta^*) & \\ mge(\tau_n = \tau_n^* : E, \sigma^*, \theta^*) \text{ if } mge(t_1 = t'_1 \dots t_n = t'_n : E, \sigma^*, \theta^*) \wedge f \equiv f' & \end{aligned}$$

To spare the need to define an extra (composition) operator for abstract substitutions, mge is defined to abstract a variant of mgu . Specifically, if $\varphi \in mgu(\{\phi(a) = \phi(b)\})$, $[\phi]_{\approx} \in \gamma_{lin}(\phi^*)$, and $mge(\{a = b\}, \phi^*, \mu^*)$, then μ^* abstracts the composition $\varphi \circ \phi$ (rather than φ), that is, $[\varphi \circ \phi]_{\approx} \in \gamma_{lin}(\mu^*)$.

To define $solve$, and thereby mge , a number of auxiliary operators are required. The first, denoted $rl(t, \phi^*)$, represents the sharing groups of ϕ^* which are relevant to the term t , that is, those sharing groups of ϕ^* which share variables with t .

Definition 15 rl [14]. The mapping $rl : Term \times Subst_{Svar}^* \rightarrow Subst_{Svar}^*$ is defined by: $rl(t, \phi^*) = \{o \in \phi^* \mid var(o) \cap var(t) \neq \emptyset\}$.

Note that $rl(t, \phi^*) = \{o \in \phi^* \mid \chi^*(t, o) \neq 0\}$ and therefore $rl(t, \phi^*) = ln(t, \phi^*) \cup nl(t, \phi^*)$. In [14] the equivalent operator is denoted rel .

The second operator, \sqcup , is a technical device which is used to calculate $occ(u, \varphi \circ \phi)$ from a set of sharing groups $occ(w, \phi)$ for the variables w with $u \in var(\varphi(w))$. Since $occ(u, \varphi \circ \phi) = \{(v, \chi_u(\varphi \circ \phi(v))) \mid u \in var(\varphi \circ \phi(v)) \wedge v \in Svar\}$, observe that $\langle v, 1 \rangle \in occ(u, \varphi \circ \phi)$ if a single variable w satisfies $u \in var(\varphi(w))$ and additionally $\chi_w(\phi(v)) = 1$ with $\chi_u(\varphi(w)) = 1$. Otherwise $\langle v, 2 \rangle \in occ(u, \varphi \circ \phi)$ if there exist distinct variables w and w' for which $u \in var(\varphi(w)) \cap var(\varphi(w'))$, or $\chi_w(\phi(v)) = 2$, or $\chi_u(\varphi(w)) = 2$. Thus $\langle v, \min(\sum_{u \in var(\varphi(w))} m_{v,w}, 2) \rangle \in occ(u, \varphi \circ \phi)$ where $m_{v,w} = \max(\chi_u(\varphi(w)), \chi_w(\phi(v)))$. The rôle of the \sqcup operator is to compute $occ(u, \varphi \circ \phi)$ by calculating the pairs $\langle v, \min(\sum_{u \in var(\varphi(w))} m_{v,w}, 2) \rangle$ given $m_{v,w}$ for $u \in var(\varphi(w))$.

Definition 16 \sqcup . The operator $\sqcup \cdot : \wp(Occ_{Svar}) \rightarrow Occ_{Svar}$ is defined by:

$$\sqcup_{w \in W} o_w = \{\langle v, \min(\sum_{(v, m_{v,w}) \in o_w} m_{v,w}, 2) \rangle \mid v \in \bigcup_{w \in W} var(o_w)\}$$

Although the motivation for \sqcup is technical, example 3 illustrates that the operator itself is straightforward to use and compute. Sometimes, for brevity, \sqcup is written infix.

Example 3. Three examples of using the \sqcup operator are given below: first, $\{\langle u, 1 \rangle, \langle v, 1 \rangle, \langle w, 2 \rangle\} \sqcup \{\langle v, 1 \rangle, \langle w, 2 \rangle, \langle x, 2 \rangle, \langle y, 1 \rangle\} = \{\langle u, \min(1, 2) \rangle, \langle v, \min(1+1, 2) \rangle, \langle w, \min(2+2, 2) \rangle, \langle x, \min(2, 2) \rangle, \langle y, \min(1, 2) \rangle\} = \{\langle u, 1 \rangle, \langle v, 2 \rangle, \langle w, 2 \rangle, \langle x, 2 \rangle, \langle y, 1 \rangle\}$; second, $\emptyset \sqcup \emptyset = \emptyset$; and third, $\sqcup_{w \in \emptyset} o_w = \emptyset$.

Note that \sqcup is commutative and associative but is not idempotent, and specifically, $o \sqcup o = var(o) \times \{2\}$. Also observe that $var(\sqcup_{w \in W} o_w) = \bigcup_{w \in W} var(o_w)$ hinting at the fact that \sqcup generalises set union which is used to combine sharing groups in the original sharing analyses [14, 21].

In the conventional approach, worst-case aliasing is always assumed and a closure under union operator is used to enumerate all the possible sharing groups that can possibly arise in unification [14, 21]. The \sqcup operator defines an analog of closure under union, closure under \sqcup , denoted $\phi^{*\star}$ and defined in definition 17.

Definition 17 closure under \sqcup , \star . The closure under \sqcup operator $\cdot^\star : Subst_{Svar}^* \rightarrow Subst_{Svar}^*$ is defined by: $\phi^{*\star} = \phi^* \cup \{o \sqcup o' \mid o, o' \in \phi^{*\star}\}$.

Closure under \sqcup is used more conservatively than the closure under union operator of [14, 21] and is only invoked in the absence of useful linearity information. An interesting consequence of $Subst_{Svar}^*(\leq)$ being a preorder (rather than a poset), is that equivalent $\phi^{*\star}$ can have different representations. For instance, if $\phi^* = \{\{(u, 1), (v, 2)\}\}$, $\phi^{*\star} = \{\{(u, 1), (v, 2)\}, \{(u, 2), (v, 2)\}\}$ but $\varphi^{*\star} \leq \phi^{*\star} \leq \varphi^{*\star}$ where $\varphi^* = \{\{(u, 2), (v, 2)\}\}$ and $\varphi^{*\star} = \{\{(u, 2), (v, 2)\}\}$. Clearly $\varphi^{*\star}$ is preferable to $\phi^{*\star}$, and more generally, redundancy can be avoided in the calculation and representation of $\phi^{*\star}$ by computing $\phi^{*\star}$ with $\{var(o) \times \{2\} \mid o \in \phi^*\}^*$.

Finally, to achieve a succinct definition of the abstract equation solver, it is useful to lift \sqcup to sets of sharing groups in the manner prescribed in definition 18.

Definition 18 \square . The mapping $\cdot \square \cdot : Subst_{Svar}^* \times Subst_{Svar}^* \rightarrow Subst_{Svar}^*$ is defined by: $\phi^* \square \phi'^* = \{o \sqcup o' \mid o \in \phi^* \wedge o' \in \phi'^*\}$.

The nub of the equation solver *mge* is *solve*. In essence, $solve(v, t, \phi^*)$ solves the syntactic equation $v = t$ in the presence of the abstract substitution ϕ^* , returning the composition of the unifier with ϕ^* . The different cases of operator *solve* apply different analysis strategies corresponding to when $\phi(v)$ is linear, $\phi(t)$ is linear, both $\phi(v)$ and $\phi(t)$ are possibly non-linear. (If both $\phi(v)$ and $\phi(t)$ are linear, cases 1 and 2 coincide.) The default strategy corresponds to the standard treatment of the abstract solver *amgu* of [14].

Definition 19 *solve*. The abstract equation solver *solve* : $Uvar \times Term \times Subst_{Svar}^* \rightarrow Subst_{Svar}^*$ is defined by:

$$solve(v, t, \phi^*) = \phi^* \setminus (rl(v, \phi^*) \cup rl(t, \phi^*)) \cup$$

$$\begin{cases} (ln(v, \phi^*) \square ln(t, \phi^*)) \cup (ln(v, \phi^*)^* \square nl(t, \phi^*)) & \text{if } nl(v, \phi^*) = \emptyset \wedge \\ & ln(v, \phi^*) \cap rl(t, \phi^*) = \emptyset \\ (ln(v, \phi^*) \square ln(t, \phi^*)) \cup (nl(v, \phi^*) \square ln(t, \phi^*)^*) & \text{if } nl(t, \phi^*) = \emptyset \wedge \\ & ln(t, \phi^*) \cap rl(v, \phi^*) = \emptyset \\ rl(v, \phi^*)^* \square rl(t, \phi^*)^* & \text{otherwise} \end{cases}$$

Note that $\phi^* \square \emptyset = \emptyset$ and $\emptyset \square \phi^* = \emptyset$ and in particular, for case 1 of *solve*, the closure $ln(v, \phi^*)^*$ need not be calculated if $nl(t, \phi^*) = \emptyset$. Similarly, in case 2, if $nl(v, \phi^*) = \emptyset$, $ln(t, \phi^*)^*$ need not be computed. The correctness of *solve* is asserted by lemma 20. The justification of lemma 20 relies on very weak properties of substitutions, and specifically, only that a most general unifier, if it exists, is idempotent.

Lemma 20.

$$\begin{aligned} [\phi]_z &\in \gamma_{lin}(\phi^*) \wedge \varphi \in mgu(\{\phi(v) = \phi(t)\}) \wedge \\ \{v\} \cup var(t) &\subseteq Svar \wedge v \notin var(t) \Rightarrow [\varphi \circ \phi]_z \in \gamma_{lin}(solve(v, t, \phi^*)) \end{aligned}$$

The correctness of *mge* follows from lemma 20 and is stated as corollary 21.

Corollary 21.

$$[\phi]_{\approx} \in \gamma_{lin}(\phi^*) \wedge \varphi \in mgu(\phi(E)) \wedge \\ mge(E, \phi^*, \mu^*) \wedge var(E) \subseteq Svar \Rightarrow [\varphi \circ \phi]_{\approx} \in \gamma_{lin}(\mu^*)$$

It is convenient to regard mge as a mapping, that is, $mge(E, \phi^*) = \mu^*$ if $mge(E, \phi^*, \mu^*)$. Strictly, it is necessary to show that $mge(E, \phi^*, \mu^*)$ is deterministic for $mge(E, \phi^*)$ to be well-defined. Like in [5], the conjecture is that mge yields a unique abstract substitution regardless of the order in which E is solved. This conjecture, however, is only really of theoretical interest because all that really matters is that any abstract substitution derived by mge is safe. This is essentially what corollary 21 asserts.

To define $unify^*$, the finite analog of $unify^c$, it is necessary to introduce an abstract restriction operator, denoted $\cdot \upharpoonright^* \cdot$.

Definition 22 abstract restriction, \upharpoonright^* . The abstract restriction operator $\cdot \upharpoonright^* \cdot : Subst_{Svar}^* \times \wp(Uvar) \rightarrow Subst_{Svar}^*$ is defined by: $\phi^* \upharpoonright^* U = \{o \upharpoonright^* U \mid o \in \phi^*\}$ where $o \upharpoonright^* U = \{(u, m) \in o \mid u \in U\}$.

The definition of $unify^*$ is finally given below, followed by the local safety theorem, theorem 24.

Definition 23 $unify^*$. The mapping $unify^* : Atom \times Subst_{Pvar}^* \times Atom \times Subst_{Pvar}^* \rightarrow Subst_{Pvar}^*$ is defined by:

$$unify^*(a, \phi^*, b, \psi^*) = mge(\{a = \Upsilon(b)\}, \phi^* \cup \Upsilon(\psi^*)) \upharpoonright^* Pvar$$

Theorem 24 local safety of $unify^*$.

$$\Phi \subseteq \gamma_{lin}(\phi^*) \wedge \Psi \subseteq \gamma_{lin}(\psi^*) \wedge \\ var(a) \cup var(b) \subseteq Pvar \Rightarrow unify^c(a, \Phi, b, \Psi) \subseteq \gamma_{lin}(unify^*(a, \phi^*, b, \psi^*))$$

Examples 4 and 5 demonstrate the precision in propagating groundness information that the domain inherits from sharing groups, and accuracy that is additionally obtained by tracking linearity. Furthermore, example 6 illustrates that the domain is more powerful than the sum of its parts, that is, it can trace linearity and sharing better than is achievable by running the Søndergaard [22, 5] and sharing group analyses [14, 21] together in lock step [6]. The examples also comment on the efficiency of the analysis.

Example 4 propagating groundness. The supremacy of the sharing group domains over the Søndergaard domain for propagating groundness information can be illustrated by separately solving two equations, first, $x = f(y, z)$ and second, $x = f(g, g)$. Suppose $Svar = \{x, y, z\}$. To demonstrate the groundness propagation of sharing groups, let $\phi^* = \{\emptyset, \{(x, 2)\}, \{(y, 2)\}, \{(z, 2)\}\}$ so that worst-case linearity is assumed. Solving $x = f(y, z)$ for ϕ^* yields

$$\varphi^* = solve(x, f(y, z), \phi^*) = \\ \{\emptyset, \{(x, 2), (y, 2)\}, \{(x, 2), (z, 2)\}, \{(x, 2), (y, 2), (z, 2)\}\}$$

Since x occurs in each (non-empty) sharing group of φ^* , grounding x must also ground both y and z , and indeed $\psi^* = \text{solve}(x, f(g, g), \phi^*) = \{\emptyset\}$. Furthermore, ψ^* indicates that y and z are independent. In contrast, the abstract unification algorithm proposed for the Søndergaard domain [5], cannot infer that x and y are grounded or independent.

Example 5 tracking linearity. Suppose $E = \{x = u, y = f(u, v), z = v\}$ and consider the abstraction of $mgu(E)$ and specifically the calculation $mge(E, \text{lin}([\epsilon]_{\approx}))$. Assuming $Svar = \{u, v, x, y, z\}$, dubbing $\epsilon^* = \text{lin}([\epsilon]_{\approx}) = \{\emptyset, \{\langle u, 1 \rangle\}, \{\langle v, 1 \rangle\}, \{\langle x, 1 \rangle\}, \{\langle y, 1 \rangle\}, \{\langle z, 1 \rangle\}\}$, and solving the equations left-to-right

$$\begin{aligned}\phi^* = \text{solve}(x, u, \epsilon^*) &= \{\emptyset, \{\langle u, 1 \rangle, \langle x, 1 \rangle\}, \{\langle v, 1 \rangle\}, \{\langle y, 1 \rangle\}, \{\langle z, 1 \rangle\}\} \\ \varphi^* = \text{solve}(y, f(u, v), \phi^*) &= \{\emptyset, \{\langle u, 1 \rangle, \langle x, 1 \rangle, \langle y, 1 \rangle\}, \{\langle v, 1 \rangle, \langle y, 1 \rangle\}, \{\langle z, 1 \rangle\}\} \\ \psi^* = \text{solve}(z, v, \varphi^*) &= \{\emptyset, \{\langle u, 1 \rangle, \langle x, 1 \rangle, \langle y, 1 \rangle\}, \{\langle v, 1 \rangle, \langle y, 1 \rangle, \langle z, 1 \rangle\}\}\end{aligned}$$

Therefore $\psi^* = mge(E, \epsilon^*)$ and indeed $\psi = \{x \mapsto u, y \mapsto f(u, v), z \mapsto v\} \in mgu(E)$ with $[\psi]_{\approx} \in \gamma_{\text{lin}}(\psi^*)$. Without exploiting linearity (or freeness), the sharing group analyses of [14, 21] have to include an additional sharing group $\{u, v, x, y, z\}$ for possible aliasing between u and v (and x and z). Tracking linearity strengthens the analysis, allowing it to deduce that u and v (and x and z) are definitely not aliased. Note also that the size of the data structure (the abstract substitution ψ^*) is pruned from 4 to 3 sharing groups and that, in contrast to the analyses of [14, 21], the calculation of a closure is avoided.

Example 6 refined sharing and linearity. The domain refines the way linearity information is recorded and in particular the analysis can differentiate between which variables can occur multiply in a term (or binding) and which variables always occur singly in a term (or binding). For instance, consider the set of substitutions $\Phi = \{[\phi]_{\approx}, [\phi']_{\approx}\}$ where $\phi = \{x \mapsto f(u, v)\}$ and $\phi' = \{x \mapsto f(w, w)\}$. Φ represents two possible bindings for x . In the first, $\phi(x)$ is linear, whereas in the second, $\phi'(x)$ is non-linear. This is reflected in $\phi^* = \alpha_{\text{lin}}(\Phi) = \text{lin}([\phi]_{\approx}) \cup \text{lin}([\phi']_{\approx})$, and specifically, if $Svar = \{u, v, w, x, y, z\}$

$$\phi^* = \{\emptyset, \{\langle u, 1 \rangle, \langle x, 1 \rangle\}, \{\langle v, 1 \rangle, \langle x, 1 \rangle\}, \{\langle w, 1 \rangle, \langle x, 2 \rangle\}, \{\langle y, 1 \rangle\}, \{\langle z, 1 \rangle\}\}$$

The abstraction ϕ^* indicates that u and v never occur more than once through $\phi(x)$ and $\phi'(x)$, and that w can occur multiply through $\phi(x)$ or $\phi'(x)$. Informally, the abstraction records why x is possibly non-linear. This, in turn, can lead to improved precision and efficiency, as is illustrated by the calculation of $mge(\{x = f(y, z), w = g\}, \phi^*)$. Again, solving the equations left-to-right

$$\begin{aligned}\varphi^* = \text{solve}(x, f(y, z), \phi^*) &= \{\emptyset, \{\langle u, 1 \rangle, \langle x, 1 \rangle, \langle y, 1 \rangle\}, \{\langle u, 1 \rangle, \langle x, 1 \rangle, \langle z, 1 \rangle\}, \\ &\quad \{\langle v, 1 \rangle, \langle x, 1 \rangle, \langle y, 1 \rangle\}, \{\langle v, 1 \rangle, \langle x, 1 \rangle, \langle z, 1 \rangle\}, \\ &\quad \{\langle w, 1 \rangle, \langle x, 2 \rangle, \langle y, 2 \rangle\}, \{\langle w, 1 \rangle, \langle x, 2 \rangle, \langle z, 2 \rangle\}, \\ &\quad \{\langle w, 1 \rangle, \langle x, 2 \rangle, \langle y, 2 \rangle, \langle z, 2 \rangle\}\} \\ \psi^* = \text{solve}(w, g, \varphi^*) &= \{\emptyset, \{\langle u, 1 \rangle, \langle x, 1 \rangle, \langle y, 1 \rangle\}, \{\langle u, 1 \rangle, \langle x, 1 \rangle, \langle z, 1 \rangle\}, \\ &\quad \{\langle v, 1 \rangle, \langle x, 1 \rangle, \langle y, 1 \rangle\}, \{\langle v, 1 \rangle, \langle x, 1 \rangle, \langle z, 1 \rangle\}\}\end{aligned}$$

In terms of precision, linearity is still exploited for u and v , even though worst-case aliasing has to be assumed for w . Consequently, on grounding w , u and v (and y and z) become independent. The Søndergaard domain, however, cannot resolve linearity to the same degree of accuracy and therefore the analysis of [5] cannot infer u and v (and y and z) become unaliased. Also, the combined domains approach [6] does not help, since the precision comes from restructuring the domain. In terms of efficiency, observe that although the closure of $\text{ln}(f(y, z), \phi^*)$ is computed, the number of sharing groups in φ^* is kept low by only combining $\text{ln}(f(y, z), \phi^*)^*$ with $\text{nl}(\mathbf{x}, \phi^*)$ (rather than with $\text{rl}(\mathbf{x}, \phi^*)$).

The extra expressiveness of the domain is not confined to abstracting multiple substitutions. If $\mu = \{\mathbf{x} \mapsto f(u, v, w, w)\}$ and $\mu^* = \text{lin}([\mu]_{\approx})$, for instance,

$$\mu^* = \{\emptyset, \{\langle u, 1 \rangle, \langle \mathbf{x}, 1 \rangle\}, \{\langle v, 1 \rangle, \langle \mathbf{x}, 1 \rangle\}, \{\langle w, 1 \rangle, \langle \mathbf{x}, 2 \rangle\}, \{\langle y, 1 \rangle\}, \{\langle z, 1 \rangle\}\}$$

so that μ^* is structurally identical to ϕ^* . Although omitted for brevity, the calculation $\text{mge}(\{x = f(y_1, y_1, y_3, y_4), w = g\}, \mu^*)$ deduces that y_i and y_j (for $i \neq j$) become independent after w is grounded. This, again, cannot be inferred in terms of the Søndergaard domain.

5 Related work

Recently, four interesting proposals for computing accurate sharing information have been put forward in the literature. In the first proposal [6], domains and analyses are combined to improve accuracy. This paper develops this theme and explores the virtues of fusing linearity with sharing groups. In short, this paper explains how accuracy and efficiency can be further improved by restructuring a combined domain as a single domain.

In the second proposal [4], the correctness of freeness analyses is considered. An abstract unification algorithm is proposed as a basis for constructing accurate freeness analyses with a domain formulated in terms of abstract equations. Safety follows because the abstract algorithm mimics the solved form algorithm in an intuitive way. Correctness is established likewise here. The essential distinction between the two works is that this paper tracks groundness and linearity. Consequently, the approach presented here can derive more accurate sharing information. Also, as pointed out in [2], “it is doubtful whether it (the abstract unification algorithm of [4]) can be the basis for a very efficient analysis”. The analysis presented here, on the other hand, is designed to be efficient.

Very recently, in the third proposal [2], an analysis for sharing, groundness, linearity and freeness is formalised as a transition system which reduces a set of abstract equations to an abstract solved form. Sharing is represented in a sharing group fashion with variables enriched with linearity and freeness information by an annotation mapping. The domain, however, essentially adopts the Jacobs and Langen [14] structure. Consequently the analysis cannot always derive sharing as accurately as the analysis reported here. Moreover, the use of a tightly-coupled domain seems to simplify some of the analysis machinery. For instance, the notion of abstraction introduced in this paper is more succinct than the equivalent

definition in [2]. This simplicity seems to stem from the fact the domain is an elegant and natural generalisation of sharing groups [14]. Also, the analysis of [2] has not, as yet, been proved correct.

Fourthly, a referee pointed out a freeness analysis which also tracks linearity to avoid the calculation of closures in sharing groups [11]. Interestingly, [11] seems to adopt a conventional notion of linearity, rather than embedding linearity into sharing groups in the useful way that is described in this paper.

To be fair, however, the analyses of [11, 4, 2] do infer freeness. This can be useful [13]. Although freeness information is not derived in this paper, it seems that freeness can be embedded into sharing groups in a similar way to linearity. What is more, if freeness is recorded this way, it can be used to improve sharing beyond what is achievable by just tracing linearity! This is unusual, contrasts to [2], and is further evidence for the usefulness of restructuring sharing groups.

6 Conclusions

A powerful, formally justified and potentially efficient analysis has been presented for inferring definite groundness and possible sharing between the variables of a logic program. The analysis builds on the combined domain approach [6] by elegantly representing linearity information in a sharing group format. By revising sharing groups to capture linearity, a single coherent domain and analysis has been formulated which more precisely captures aliasing behaviour; propagates groundness information with greater accuracy; and in addition, yields a more refined notion of linearity. In more pragmatic terms, the analysis permits aliasing and groundness to be inferred to a higher degree of accuracy than in previous proposals. The analysis is significant because sharing information underpins many optimisations in logic programming.

Acknowledgements

Thanks are due to Manuel Hermenegildo and Dennis Dams for useful discussions on linearity. This work was supported by ESPRIT project (6707) “ParForce”.

References

1. M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *J. Logic Programming*, 10:91–124, 1991.
2. M. Bruynooghe and M. Codish. Freeness, sharing, linearity and correctness – all at once. In *WSA'93*, pages 153–164, September 1993.
3. J.-H. Chang and A. M. Despain. Semi-intelligent backtracking of prolog based static data dependency analysis. In *JICSLP'85*. IEEE Computer Society, 1985.
4. M. Codish, D. Dams, G. Filé, and M. Bruynooghe. Freeness analysis for logic programs - and correctness? In *ICLP'93*, pages 116–131. MIT Press, June 1993.

5. M. Codish, D. Dams, and E. Yardeni. Derivation and safety of an abstract unification algorithm for groundness and aliasing analysis. In *ICLP'91*, pages 79–93, Paris, France, 1991. MIT Press.
6. M. Codish, A. Mulkers, M. Bruynooghe, M. J. García de la Banda, and M. Hermenegildo. Improving abstract interpretation by combining domains. In *PEPM'93*. ACM Press, 1993.
7. A. Cortesi and G. Filé. Abstract interpretation of logic programs: an abstract domain for groundness, sharing, freeness and compoundness analysis. In *PEPM'91*, pages 52–61. ACM Press, 1991.
8. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*, pages 238–252. ACM Press, 1977.
9. D. Dams. Personal communication on linearity lemma 2.2. July, 1993.
10. S. K. Debray. Static inference of modes and data dependencies in logic programs. *ACM TOPLAS*, 11(3):418–450, July 1989.
11. W. Hans and S. Winkler. Aliasing and groundness analysis of logic programs through abstract interpretation and its safety. Technical Report Nr. 92-27, RWTH Aachen, Lehrstuhl für Informatik II Ahornstraße 55, W-5100 Aachen, 1992.
12. M. Hermenegildo. Personal communication on freeness analysis. May, 1993.
13. M. Hermenegildo and F. Rossi. Non-strict independent and-parallelism. In *ICLP'90*, pages 237–252, Jerusalem, 1990. MIT Press.
14. D. Jacobs and A. Langen. Static Analysis of Logic Programs. *J. Logic Programming*, pages 154–314, 1992.
15. A. King. A new twist to linearity. Technical Report CSTR 93-13, Department of Electronics and Computer Science, Southampton University, Southampton, 1993.
16. J. Lassez, M. J. Maher, and K. Marriott. *Foundations of Deductive Databases and Logic Programming*, chapter Unification Revisited. Morgan Kaufmann, 1987.
17. B. Le Charlier, K. Musumbu, and P. Van Hentenryck. A generic abstract interpretation algorithm and its complexity. In *ICLP'91*, pages 64–78. MIT Press, 1991.
18. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.
19. K. Marriott and H. Søndergaard. Analysis of constraint logic programs. In *NACLP'90*, pages 531–547. MIT Press, 1990.
20. K. Muthukumar and M. Hermenegildo. Combined determination of sharing and freeness of program variables through abstract interpretation. In *ICLP'91*, pages 49–63, Paris, France, 1991. MIT Press.
21. K. Muthukumar and M. Hermenegildo. Compile-time derivation of variable dependency through abstract interpretation. *J. of Logic Programming*, pages 315–437, 1992.
22. H. Søndergaard. An application of the abstract interpretation of logic programs: occur-check reduction. In *ESOP'86*, pages 327–338. Springer-Verlag, 1986.
23. R. Sundararajan and J. Conery. An abstract interpretation scheme for groundness, freeness, and sharing analysis of logic programs. In *12th FST and TCS Conference*, New Delhi, India, December 1992. Springer-Verlag.
24. A. Taylor. *High Performance Prolog Implementation*. PhD thesis, Basser Department of Computer Science, Sydney, Australia, July 1991.
25. H. Xia. *Analyzing Data Dependencies, Detecting And-Parallelism and Optimizing Backtracking in Prolog Programs*. PhD thesis, University of Berlin, April 1989.

A π -calculus Specification of Prolog

Benjamin Z. Li

Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104-6389 USA
zhenli@saul.cis.upenn.edu

Abstract. A clear and modular specification of Prolog using the π -calculus is presented in this paper. Prolog goals are represented as π -calculus processes, and Prolog predicate definitions are translated into π -calculus agent definitions. Prolog's depth-first left-right control strategy as well as the cut control operator are modeled by the synchronized communication among processes, which is similar in spirit to continuation-passing style implementation of Prolog. Prolog terms are represented by persistent processes, while logical variables are modeled by complex processes with channels that, at various times, can be written, read, and reset. Both unifications with and without backtracking are specified by π -calculus agent definitions. A smooth merging of the specification for control and the specification for unification gives a full specification for much of Prolog. Some related and further works are also discussed.

1 Introduction

Prolog is a simple, powerful and efficient programming language, but its depth-first left-right control as well as the control operator `cut` (!) and its lack of the occurs check destroys the declarative reading of Prolog programs. For example, a left recursive clause will cause an infinite computation while a right recursive clause with same logic reading will terminate the computation. Hence, logic does not provide a simple and formal semantics for Prolog. In [Ros92a] Ross provided an interesting specification of Prolog control by mapping it into processes in the concurrent specification language CCS [Mil89]. In this paper, we develop and extend this approach significantly by using the π -calculus, a richer concurrent specification language. We are not only able to specify Prolog's control primitives but also its correct interaction with Prolog's unification, including the lack of the occur-check and the construction of circular terms.

The π -calculus [MPW92a, MPW92b, Mil91] is a calculus for modeling concurrent systems with evolving communication structure. It has been proven very powerful in modeling functional programming languages [Mil90b] and object-oriented programming languages [Wal90]. In this paper, we will introduce a clear and modular specification for Prolog using the π -calculus. The specification is modular in the sense that the Prolog control part and unification are specified separately, but can be merged together smoothly to form a full specification for Prolog. In fact, part of the motivation behind this paper was to understand how successfully the π -calculus could be used to specify the operational semantics of a non-trivial programming language, in this case Prolog. As we hope it will be clear from this paper, the π -calculus, along with a sorting discipline proposed for it, does indeed provide an attractive specification language.

The rest of this paper is organized as follows. The π -calculus will be briefly introduced in Section 2. A π -calculus specification for Prolog's depth-first left-right control as well as the cut control will be defined in Section 3, and a π -calculus specification for unification will be discussed in Section 4. Section 5 will merge these two specifications together to achieve a specification for full Prolog.

We will compare with some related works and discuss future works in Section 6. Section 7 is the conclusion.

2 The π -calculus

The π -calculus is a model of concurrent computation based upon the notation of naming, which provides an identity to an entity that allows it to concurrently coexist in an environment with other entities. The primitive elements of π -calculus are structureless entities called *Names*, infinitely many and denoted by $\{x, y, z, \dots\} \in \mathcal{N}$. A name refers to a communication channel. If the name x represents the input end of a channel, then the co-name \bar{x} represents its output end. In the following syntax of the π -calculus, P_1, P_2, P range over process expressions, A ranges over agent identifiers \mathcal{K} , and \tilde{y} is an abbreviation for a sequence of names $y_1 \dots y_n$ ($n \geq 0$).

$$P ::= 0 \mid \bar{x}\tilde{y}.P \mid x(\tilde{y}).P \mid P_1 + P_2 \mid P_1 | P_2 \mid (\nu x)P \mid [x=y]P \mid [x \neq y]P \mid A(\tilde{y}) \mid !P$$

where

- 0 is the *inaction* process which can do nothing.
- $\bar{x}\tilde{y}.P$ can output the name(s) \tilde{y} along the channel x and then becomes P .
- $x(\tilde{y}).P$ can input some arbitrary name(s) \tilde{z} along the channel x and then becomes $P\{\tilde{z}/\tilde{y}\}$. Of course, \tilde{y} and \tilde{z} have to be of the same length.
- A summation $P_1 + P_2$ can behave as either P_1 or P_2 non-deterministically.
- A composition $P_1 | P_2$ means that P_1 and P_2 are concurrently active, so they can act independently but can also communicate. For example, if $P = \bar{x}z.P'$ and $Q = x(y).Q'$ then $P|Q$ means that either P can output z along channel x ; or Q can input an name along channel x ; or P and Q can communicate internally by performing a *silent* action τ and then becomes $P'|Q'\{z/y\}$.
- A restriction $(\nu x)P$ declares a new name (private name) x in P that is different from all external names. For example, $(\nu x)(\bar{x}z.P|x(y).Q')$ can only perform internal communication, but $\bar{x}z.P|(\nu x)x(y).Q'$ can not communicate. Actually, $(\nu x)x(y).Q'$ is a dead process, which is equivalent to 0 .
- A match $[x=y]P$ behaves like P if x and y are identical, and otherwise like 0 .
- A mis-match $[x \neq y]P$ behaves like P if x and y are not identical, and otherwise like 0 .¹
- A *defined* agent $A(\tilde{y})$ must have a corresponding defining equation of the form: $A(\tilde{x}) \stackrel{\text{def}}{=} P$. Then $A(\tilde{y})$ is the same as $P\{\tilde{y}/\tilde{x}\}$.
- A replication $!P$, which can be defined as: $!P \stackrel{\text{def}}{=} P|!P$, provides infinite copies of P in composition, i.e. $!P = P|P\dots$

A *transition* in the π -calculus is of the form: $P \xrightarrow{\alpha} Q$, which means that P can evolve into Q by performing the action α . The action α can be one of the $\tau, \bar{x}\tilde{y}, x(\tilde{y})$ and a fourth action called *bound output action* which allows a process to output a private name and hence widen the scope of the private name. The formal definition of translation relation $\xrightarrow{\alpha}$ is given in [MPW92b]. Here is a simple

¹ the mis-match is not presented in the original π -calculus, but is included here to simplify the specifications presented in this paper.

example:

$$\begin{aligned}
 (a.P_1 + b.\bar{c}.P_2) \mid x(y).\bar{y}.Q \mid \bar{x}b.R &\xrightarrow{\tau} (a.P_1 + b.\bar{c}.P_2) \mid \bar{b}.Q\{b/y\} \mid R \\
 &\xrightarrow{\tau} \bar{c}.P_2 \mid Q\{b/y\} \mid R \\
 &\xrightarrow{\bar{c}} P_2 \mid Q\{b/y\} \mid R
 \end{aligned}$$

Some convenient abbreviations are used in this paper, such as $x(y)$ (resp. $\bar{x}y$) as an abbreviation for $x(y).0$ (resp. $\bar{x}y.0$), $(\nu x_1 \dots x_n)P$ for $(\nu x_1) \dots (\nu x_n)P$, and $\xrightarrow{\alpha_1 \dots \alpha_n}$ for n sequential transitions $\xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_n}$. We also use the anonymous channel name $_-$ when the name itself does not matter.

Names in \tilde{y} are said to be bound in $x(\tilde{y}).P$, and so is the name y in $(vy)P$. Some processes are considered to be equivalent according to the *structural congruence* \equiv relation, defined in [Mil91]. Two processes are *observation-equivalent* or *bisimilar* (\approx) if their input/output relationships (ignoring the internal communication τ) are the same. Bisimilarity and process equivalence are discussed in [MPW92b, Mil89].

3 Specifying the Prolog Control in the π -calculus

In this section, we will specify the Prolog's depth-first left-right control strategy and the cut control operator in the π -calculus. For simplicity, Prolog terms and unification will be ignored until next section. A Prolog goal is represented as a π -calculus process, and a Prolog predicate definition is translated into a π -calculus agent definition.

3.1 Prolog Goals as Processes

The evaluation of a Prolog goal G can result in either *success* or *fail*. A successfully evaluated goal G might be *backtracked* to find alternative solutions. So the corresponding π -calculus process $\llbracket G \rrbracket(s, f, b)$ is associated with three channel names: s (for *success* channel), f (for *fail* channel), b (for *backtracking* channel). Its behavior can be described as follows:

$$\llbracket G \rrbracket(s, f, b) \approx \begin{cases} \bar{s}.b.G_{alt_sol}(s, f, b) & \text{if evaluation of the Goal } G \text{ succeeds.} \\ f & \text{if evaluation of the Goal } G \text{ fails.} \end{cases}$$

After having found one solution, $\llbracket G \rrbracket(s, f, b)$ sends an output action \bar{s} and then waits on the *backtracking* channel b before computing alternative solutions (denoted by $G_{alt_sol}(s, f, b)$). Suppose a goal G can produce n ($n \geq 0$) solutions, then $\llbracket G \rrbracket(s, f, b) \approx (\bar{s}.b.)^n f$.

A left-associative sequential-and control operator \triangleright is introduced in order to simplify the notation of the corresponding process for a Prolog conjunctive goal (P, Q) ²:

$$(A \triangleright B)(s, f, b) \triangleq (\nu s' f')(A(s', f, f') \mid !s'.B(s, f', b)) \quad (1)$$

$$\llbracket (P, Q) \rrbracket(s, f, b) \triangleq (\llbracket P \rrbracket \triangleright \llbracket Q \rrbracket)(s, f, b) \quad (2)$$

$$\equiv (\nu s' f')(\llbracket P \rrbracket(s', f, f') \mid !s'.\llbracket Q \rrbracket(s, f', b)) \quad (3)$$

The behaviors of definition (3) can be understood as follows:

² \triangleq is used for translations from Prolog to π -calculus while $\stackrel{def}{=}$ is used in π -calculus agent definitions.

- If $\llbracket P \rrbracket(s', f, f')$ reports *fail* via \overline{f} , so does $\llbracket P, Q \rrbracket(s, f, b)$ since they use the same channel f .
- If $\llbracket P \rrbracket(s', f, f')$ reports *success* via $\overline{s'}$, then one copy of $\llbracket Q \rrbracket(s, f', b)$ will be activated after the synchronized communication of $\overline{s'}$ and s' . And then,
 - If $\llbracket Q \rrbracket(s, f', b)$ reports *success* via \overline{s} , so does $\llbracket P, Q \rrbracket(s, f, b)$ since both use the same channel s .
 - If $\llbracket Q \rrbracket(s, f', b)$ reports *fail* via $\overline{f'}$, then $\llbracket P \rrbracket(s', f, f')$ will be backtracked.

The bang ! before $s'.$ $\llbracket Q \rrbracket(s, f', b)$ is necessary because backtracked $\llbracket P \rrbracket(s', f, f')$ may find another solution and then need to invoke $\llbracket Q \rrbracket(s, f', b)$ again.

Similarly, a left-associative sequential-or control operator \oplus is used for disjunctive goal $(P; Q)$:

$$(A \oplus B)(s, f, b) \stackrel{\Delta}{=} (\nu f')(A(s, f', b) \mid f'.B(s, f, b)) \quad (4)$$

$$\begin{aligned} \llbracket (P; Q) \rrbracket(s, f, b) &\stackrel{\Delta}{=} (\llbracket P \rrbracket \oplus \llbracket Q \rrbracket)(s, f, b) \\ &\equiv (\nu f')(\llbracket P \rrbracket(s, f', b) \mid f'.\llbracket Q \rrbracket(s, f, b)) \end{aligned} \quad (5)$$

where $\llbracket P \rrbracket$ and $\llbracket Q \rrbracket$ use the same s and b channels, corresponding to the **or** relationship of P and Q . However, $\llbracket Q \rrbracket$ can be activated only after $\llbracket P \rrbracket$ reports *fail* via $\overline{f'}$. Thus, we exactly models Prolog's sequential nature of **or** and Prolog's sequential clause searching as we shall see in Section 3.2.

Using the *sort* notation in [Mil91], we introduce two *sorts*, **Succ** and **Fail** for the *success* channel names and *fail* channel names respectively. Based on the above discussion, it is clear that *backtracking* channel names have the same sort as the *fail* channel names. The following sorting:

$$\{\mathbf{Succ} \mapsto (), \mathbf{Fail} \mapsto ()\}$$

means that both *success* channels and *fail* channels carry no names. A general sorting of the form $S \mapsto (S_1, \dots, S_n)$ means that along a channel s of sort S , we can receive or send n names of sorts S_1, \dots, S_n respectively. We also use $s \hat{t} t$ for the concatenation of sorts, e.g. $(S_1) \hat{t} (S_2, S_3) = (S_1, S_2, S_3)$. The following notations are used to specify sorts for names and agents:

- $s, s_i, s' : \mathbf{Succ}$ means that names s, s_i, s' are of sort **Succ**. And similarly, $f, b, f_i, f' : \mathbf{Fail}$ means that f, b, f_i, f' are of sort **Fail**.
- $\llbracket G \rrbracket, \llbracket P, Q \rrbracket, \llbracket P; Q \rrbracket : (\mathbf{Succ}, \mathbf{Fail}, \mathbf{Fail})$ means that agents $\llbracket G \rrbracket, \llbracket P, Q \rrbracket, \llbracket P; Q \rrbracket$ shall take three name arguments of sorts **Succ**, **Fail**, **Fail** respectively.

3.2 Prolog Predicate Definitions as π -calculus Agent Definitions

A Prolog predicate is defined by a sequence of clauses. Suppose we have the following definition for a predicate P , consisting of m clauses:

$$P = \begin{cases} P :- Body_1. \\ P :- Body_2. \\ \dots \\ P :- Body_m. \end{cases}$$

Then the corresponding π -calculus agent definition for $P : (\mathbf{Succ}, \mathbf{Fail}, \mathbf{Fail})$ is:

$$\begin{aligned} P(s, f, b) &\stackrel{\text{def}}{=} (P_1 \oplus P_2 \oplus P_3 \oplus \dots \oplus P_m)(s, f, b) \\ &\equiv (\nu f_1 f_2 \dots f_{m-1})(P_1(s, f_1, b) \mid f_1.P_2(s, f_2, b) \mid f_2.P_3(s, f_3, b) \mid \\ &\quad \dots \mid f_{m-1}.P_m(s, f, b)) \end{aligned} \quad (6)$$

where the definition for each P_i ($1 \leq i \leq m$) is determined by the i th clause of the predicate P .

Suppose the i th clause of P is defined as: ' $P : -B_1, B_2, \dots, B_n.$ ' , then the agent $P_i : (\text{Succ}, \text{Fail}, \text{Fail})$ is defined as follows:

$$P_i(s, f, b) \stackrel{\text{def}}{=} ([\![B_1]\!] \triangleright [\![B_2]\!] \triangleright [\![B_3]\!] \triangleright \dots \triangleright [\![B_n]\!])(s, f, b) \quad (7)$$

$$\equiv (\nu s_1 \dots s_{n-1} f_1 \dots f_{n-1})([\![B_1]\!](s_1, f, f_1) \mid !s_1.[\![B_2]\!](s_2, f_1, f_2)$$

$$\mid !s_2.[\![B_3]\!](s_3, f_2, f_3) \mid \dots \mid !s_{n-1}.[\![B_n]\!](s, f_{n-1}, b)) \quad (8)$$

The behavior of above definition can be described in a similar way as the behavior of definition (3). We can also imagine that there exist two chains in definition (8): a *success chain* connected by s_i 's, and a *backtracking chain* connected by f_i 's, as illustrated in Fig. 1.

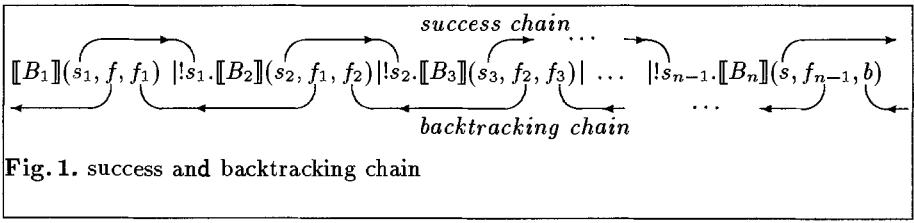


Fig. 1. success and backtracking chain

If the i th clause of P is a fact (i.e. a clause with empty body): ' $P!$ ' , then the agent P_i is defined as: $P_i(s, f, b) \stackrel{\text{def}}{=} \bar{s}.b.\bar{f}$, which says that P_i reports *success* via \bar{s} only once and then reports *fail* via \bar{f} upon receiving *backtrack* via b .

Now, let us look at a propositional Prolog example and its π -calculus translation:

$$\begin{aligned} a. \quad & A(s, f, b) \stackrel{\text{def}}{=} \bar{s}.b.\bar{f} \\ b :- a. \quad & B_1(s, f, b) \stackrel{\text{def}}{=} A(s, f, b) \quad (\text{i.e. } \equiv \bar{s}.b.\bar{f}) \\ b :- b, a. \quad & B_2(s, f, b) \stackrel{\text{def}}{=} (B \triangleright A)(s, f, b) \\ & \equiv (\nu s_1 f_1)(B(s_1, f, f_1) \mid !s_1.A(s, f_1, b)) \\ B(s, f, b) \stackrel{\text{def}}{=} & (B_1 \oplus B_2)(s, f, b) \\ & \equiv (\nu f')(B_1(s, f', b) \mid f'.B_2(s, f, b)) \end{aligned}$$

Using the structural congruence relation and transition relation discussed in Section 2, the process $B(s, f, b)$ behaves as follows:

$$\begin{aligned} B(s, f, b) \equiv & (\nu f')(B_1(s, f', b) \mid f'.B_2(s, f, b)) \equiv (\nu f')(A(s, f', b) \mid f'.B_2(s, f, b)) \\ \equiv & (\nu f')(\bar{s}.b.\bar{f}' \mid f'.B_2(s, f, b)) \xrightarrow{\bar{s}.b} (\nu f')(\bar{f}' \mid f'.B_2(s, f, b)) \\ \xrightarrow{\tau} & (0 \mid B_2(s, f, b)) \equiv B_2(s, f, b) \equiv (\nu s_1 f_1)(B(s_1, f, f_1) \mid !s_1.A(s, f_1, b)) \\ \equiv & (\nu s_1 f_1)((\nu f')(B_1(s_1, f', f_1) \mid f'.B_2(s_1, f, f_1)) \mid !s_1.A(s, f_1, b)) \\ \equiv & (\nu s_1 f_1)((\nu f')(\bar{s}_1.f_1.\bar{f}' \mid f'.B_2(s_1, f, f_1)) \mid !s_1.A(s, f_1, b)) \\ \xrightarrow{\tau} & (\nu s_1 f_1)((\nu f')(f_1.\bar{f}' \mid f'.B_2(s_1, f, f_1)) \mid A(s, f_1, b) \mid !s_1.A(s, f_1, b)) \\ \equiv & (\nu s_1 f_1)((\nu f')(f_1.\bar{f}' \mid f'.B_2(s_1, f, f_1)) \mid \bar{s}.b.\bar{f}_1 \mid !s_1.A(s, f_1, b)) \\ \xrightarrow{\bar{s}.b} & (\nu s_1 f_1)((\nu f')(f_1.\bar{f}' \mid f'.B_2(s_1, f, f_1)) \mid \bar{f}_1 \mid !s_1.A(s, f_1, b)) \\ \xrightarrow{\tau, \tau} & (\nu s_1 f_1)(B_2(s_1, f, f_1)) \mid !s_1.A(s, f_1, b) \rightarrow \dots \end{aligned}$$

which shows that $B(s, f, b)$ can perform infinitely many $\bar{s}.b.$ actions, reflecting the infinitely many solutions of predicate b .

3.3 Specifying the Cut

In Prolog, the cut $!$ is a non-declarative control operator used to cut search space. It can also be treated as a special goal, which is translated into a π -calculus agent $Cut : (\text{Fail})^\wedge(\text{Succ}, \text{Fail}, \text{Fail})$:

$$Cut(f_0)(s, f, b) \stackrel{\text{def}}{=} \bar{s}.b.\bar{f}_0 \quad (9)$$

The process $Cut(f_0)(s, f, b)$, when activated, will report *success* on \bar{s} as usually, but will reports *fail* on \bar{f}_0 instead of the regular *fail* channel (\bar{f}). As we shall see in definition (11), the f_0 must be the *fail* channel of the calling process. Suppose that the i th clause of the predicate P contains a cut as the j th goal in the body:

$$P :- B_1 \dots B_{j-1}, !, B_{j+1}, \dots, B_n.$$

Then the agent $P_i : (\text{Fail})^\wedge(\text{Succ}, \text{Fail}, \text{Fail})$ and $P : (\text{Succ}, \text{Fail}, \text{Fail})$ are defined as follows:

$$\begin{aligned} P(s, f, b) &\stackrel{\text{def}}{=} (P_1 \oplus P_2 \oplus \dots \oplus P_i(f) \oplus \dots \oplus P_m)(s, f, b) \\ &\equiv (\nu f_1 \dots f_{m-1})(P_1(s, f_1, b) | f_1.P_2(s, f_2, b) | \dots | \\ &\quad f_{i-1}.P_i(f)(s, f_i, b) | \dots | f_{m-1}.P_m(s, f, b)) \end{aligned} \quad (10)$$

$$\begin{aligned} P_i(f_0)(s, f, b) &\stackrel{\text{def}}{=} ([B_1] \triangleright \dots [B_{j-1}] \triangleright Cut(f_0) \triangleright [B_{j+1}] \triangleright \dots [B_n])(s, f, b) \quad (11) \\ &\equiv (\nu s_1 \dots s_{n-1} f_1 \dots f_{n-1})([B_1](s_1, f, f_1) | !s_1.[B_2](s_2, f_1, f_2) | \\ &\quad \dots | !s_{i-2}.[B_{i-1}](s_{i-1}, f_{i-2}, f_{i-1}) | !s_{i-1}.Cut(f_0)(s_i, f_{i-1}, f_i) | \\ &\quad !s_i.B_{i+1}(s_{i+1}, f_i, f_{i+1}) | \dots | !s_{n-1}.[B_n](s, f_{n-1}, b)) \quad (12) \end{aligned}$$

In (10), the *fail* channel f of a calling process is passed to P_i , and then is passed to the *Cut* process in (11). By identifying the failure of *Cut* (upon backtracking) with the failure of the calling process P , we achieve the effect of the cut.

Since a backtracking never passes across a cut, the $!$ before s_{i-1} and s_i in definition (12) is not necessary. So we can optimize the definitions using an optimized sequential-and operator $\triangleright|$:

$$(A \triangleright B)(s, f, b) \stackrel{\Delta}{=} (\nu s' f')(A(s', f, f') | s'.B(s, f', b)) \quad (13)$$

$$\begin{aligned} P_i(f_0)(s, f, b) &\stackrel{\text{def}}{=} ([B_1] \triangleright \dots [B_{j-1}] \triangleright Cut(f_0) \triangleright [B_{j+1}] \triangleright \dots [B_n])(s, f, b) \quad (14) \\ &\equiv (\nu s_1 \dots s_{n-1} f_1 \dots f_{n-1})([B_1](s_1, f, f_1) | !s_1.[B_2](s_2, f_1, f_2) | \\ &\quad \dots | !s_{i-2}.[B_{i-1}](s_{i-1}, f_{i-2}, f_{i-1}) | s_{i-1}.Cut(f_0)(s_i, f_{i-1}, f_i) | \\ &\quad s_i.B_{i+1}(s_{i+1}, f_i, f_{i+1}) | \dots | !s_{n-1}.[B_n](s, f_{n-1}, b)) \quad (15) \end{aligned}$$

If the second clause for the predicate b in the example of Section 3.2 is replaced with ' $b :- b, !, a.$ ', then we have:

$$\begin{aligned} A(s, f, b) &\stackrel{\text{def}}{=} \bar{s}.b.\bar{f} \\ B_1(s, f, b) &\stackrel{\text{def}}{=} A(s, f, b) \quad (\equiv \bar{s}.b.\bar{f}) \\ B_2(f_0)(s, f, b) &\stackrel{\text{def}}{=} (B \triangleright Cut(f_0) \triangleright A)(s, f, b) \\ &\equiv (\nu s_1 s_2 f_1 f_2)(B(s_1, f, f_1) | s_1.Cut(f_0)(s_2, f_1, f_2) | s_2.A(s, f_2, b)) \\ B(s, f, b) &\stackrel{\text{def}}{=} (B_1 \oplus B_2)(s, f, b) \equiv (\nu f')(B_1(s, f', b) | f'.B_2(f)(s, f, b)) \end{aligned}$$

Now, the behavior of process $B(s, f, b)$ is different from the one in Section 3.2:

$$\begin{aligned}
 B(s, f, b) &\equiv (\nu f')(B_1(s, f', b) \mid f'.B_2(f)(s, f, b)) \equiv (\nu f')(\bar{s}.b.\bar{f}' \mid f'.B_2(f)(s, f, b)) \\
 &\xrightarrow{\bar{s}, b} (\nu f')(\bar{f}' \mid f'.B_2(f)(s, f, b)) \xrightarrow{\tau} B_2(f)(s, f, b) \\
 &\equiv (\nu s_1 s_2 f_1 f_2)(B(s_1, f, f_1) \mid s_1.Cut(f)(s_2, f_1, f_2) \mid s_2.A(s, f_2, b)) \\
 &\equiv (\nu s_1 s_2 f_1 f_2)((\nu f')(\bar{s}_1.f_1.\bar{f}' \mid f'.B_2(f)(s_1, f, f_1)) \mid \\
 &\quad s_1.Cut(f)(s_2, f_1, f_2) \mid s_2.A(s, f_2, b)) \\
 &\xrightarrow{\tau} (\nu s_1 s_2 f_1 f_2)((\nu f')(f_1.\bar{f}' \mid f'.B_2(f)(s_1, f, f_1)) \mid \\
 &\quad Cut(f)(s_2, f_1, f_2) \mid s_2.A(s, f_2, b)) \\
 &\equiv (\nu s_1 s_2 f_1 f_2)((\nu f')(f_1.\bar{f}' \mid f'.B_2(f)(s_1, f, f_1)) \mid \bar{s}_2.f_2.\bar{f} \mid s_2.A(s, f_2, b)) \\
 &\xrightarrow{\tau} (\nu s_1 s_2 f_1 f_2)((\nu f')(f_1.\bar{f}' \mid f'.B_2(f)(s_1, f, f_1)) \mid f_2.\bar{f} \mid A(s, f_2, b)) \\
 &\equiv (\nu s_1 s_2 f_1 f_2)((\nu f')(f_1.\bar{f}' \mid f'.B_2(f)(s_1, f, f_1)) \mid f_2.\bar{f} \mid \bar{s}.b.\bar{f}_2) \\
 &\xrightarrow{\bar{s}, b} (\nu s_1 s_2 f_1 f_2)((\nu f')(f_1.\bar{f}' \mid f'.B_2(f)(s_1, f, f_1)) \mid f_2.\bar{f} \mid \bar{f}_2) \\
 &\xrightarrow{\tau, \bar{f}} (\nu s_1 s_2 f_1 f_2)((\nu f')(f_1.\bar{f}' \mid f'.B_2(f)(s_1, f, f_1))) \approx 0
 \end{aligned}$$

which shows that after reporting twice *success* via \bar{s} (corresponding to the two solutions of the predicate b), $B(s, f, b)$ will terminate.

4 Specifying Unification in the π -calculus

In this section, we will show how to represent Prolog terms as π -calculus processes, and logical variables as complex processes with channels that, at various times, can be written, read, and reset. We then show how to specify, in π -calculus, unification with and without backtracking.

4.1 Terms as Processes

The following sorting will be used for specifying terms:

$$\{\text{Tag} \mapsto (), \text{Cell} \mapsto (\text{Tag}, \text{Ptr} \cup \text{Cell}), \text{Ptr} \mapsto (\text{Cell}, \text{Ptr})\}$$

A channel $x:\text{Cell}$ is used for representing terms, and different kinds of terms are distinguished by different names of sort **Tag** that x carries:

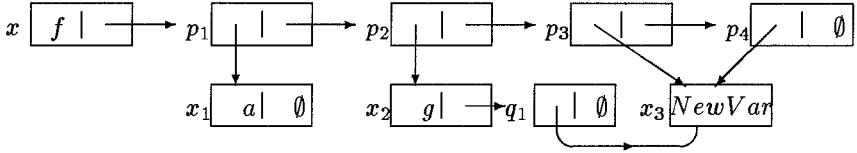
- **Var** : **Tag** denotes an unbound logical variable. **Ref** : **Tag** denotes a bound logical variable. When the first name carried on a channel $x : \text{Cell}$ is **Ref**, then the second name must be of sort **Cell**. In all other cases, the second name must be of sort **Ptr**.
- $f, g, k : \text{Tag}$ denote function or constant terms where f, g, k are functor names or constant names. As stated above, the second name that carried on x must be of sort **Ptr**.

A channel $p : \text{Ptr}$ is used for representing, in a form similar to a link list structure (see the coming example), the arguments of a function. $\emptyset : \text{Ptr}$ denotes the end of a list. The following is the translation from Prolog terms to π -calculus processes:

$$\begin{aligned}
 \llbracket f(t_1, \dots, t_n) \rrbracket(x) &\stackrel{\Delta}{=} (\nu p_1 \dots p_n x_1 \dots x_n) (!\bar{x} f p_1 \mid !\bar{p_1} x_1 p_2 \mid !\bar{p_2} x_2 p_3 \mid \dots \\
 &\quad |\!\bar{p_n} x_n \emptyset| \llbracket t_1 \rrbracket(x_1) \mid \dots \mid \llbracket t_n \rrbracket(x_n)) \\
 \llbracket k \rrbracket(x) &\stackrel{\Delta}{=} !\bar{x} k \emptyset \\
 \llbracket X \rrbracket(x) &\stackrel{\Delta}{=} \text{NewVar}(x) \text{ Defined in Fig. 2}
 \end{aligned}$$

with the restriction that several occurrences of a same variable X must use the same channel x and such $\llbracket X \rrbracket(x)$ must be translated only once, e.g. the three occurrences of X in the following example. The picture shows the 'link list' representation of the arguments of $f(a, g(X), X, X)$.

$$\begin{aligned} \llbracket f(a, g(X), X, X) \rrbracket(x) &\stackrel{\Delta}{=} (\nu p_1 p_2 p_3 p_4 x_1 x_2 x_3)(\overline{x} f p_1 \\ &\quad | \overline{p_1} x_1 p_2 | \overline{p_2} x_2 p_3 | \overline{p_3} x_3 p_4 | \overline{p_4} x_3 \emptyset \\ &\quad | \overline{x_1} a \emptyset | (\nu q_1)(\overline{x_2(g q_1)} | \overline{q_1} x_3 \emptyset) | NewVar(x_3)) \end{aligned}$$



4.2 Logical Variable and Pure Unification

A definition of the agent $NewVar$ and a specification of pure unification without backtracking is illustrated in Fig. 2, where a logical variable is represented by

$$\begin{aligned} NewVar(x) &\stackrel{\text{def}}{=} \overline{x} \text{ Var } \emptyset . NewVar(x) + x(-y) . BndVar(x, y) \\ BndVar(x, y) &\stackrel{\text{def}}{=} \overline{x} \text{ Ref } y \\ Dref(x, r) &\stackrel{\text{def}}{=} x(\text{tag } y) . ([\text{tag} = \text{Ref}] Dref(y, r) + [\text{tag} \neq \text{Ref}] \overline{r} x) \\ (x=_u y)(s, f, b) &\stackrel{\text{def}}{=} (\nu r_x r_y) (Dref(x, r_x) | Dref(y, r_y) | r_x(x_1) . r_y(y_1) . \\ &\quad ([x_1 = y_1] \overline{s} . b . \overline{f} + \\ &\quad [x_1 \neq y_1] x_1(\text{tag}_x p_x) . y_1(\text{tag}_y p_y) . \\ &\quad ([\text{tag}_x = \text{Var}] \overline{x_1} - y_1 . \overline{s} . b . \overline{f} + \\ &\quad [\text{tag}_y = \text{Var}] \overline{y_1} - x_1 . \overline{s} . b . \overline{f} + \\ &\quad [\text{tag}_x \neq \text{Var}] [\text{tag}_y \neq \text{Var}] ([\text{tag}_x = \text{tag}_y] (p_x =_p p_y) (s, f, b) + \\ &\quad [\text{tag}_x \neq \text{tag}_y] \overline{f})))) \\ (p=_p q)(s, f, b) &\stackrel{\text{def}}{=} [p = q] \overline{s} . b . \overline{f} + \\ &\quad [p \neq q] p(x p_1) . q(y q_1) . ((x =_u y) \triangleright (p_1 =_p q_1))(s, f, b) \end{aligned}$$

Fig. 2. Pure Unification Without Backtracking

a process with two states:

- the unbound state $NewVar(x)$ which can either send (**Var** \emptyset) along channel x to indicate that x is not bound yet, or receive ($-y$) on channel x to indicate that x is going to be bound to the cell y , and therefore enter the bound state $BndVar(x, y)$.
- the bound state $BndVar(x, y)$ which always send (**Ref** y) along channel x to indicate that x is bound to y .

When x is bound to y in the state $BndVar(x, y)$, y itself may denote an unbound logical variable and may be bound to another cell later. Hence, possible reference chains can exist, such as in Fig. 3. In order to unify two cells

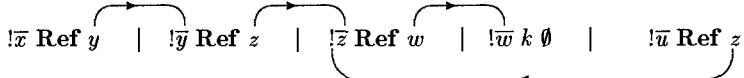


Fig. 3. Reference Chain

(representing two terms), *dereferencing* is performed by a process $Dref(x, r)$ which, when applied to a cell x , will send along channel r the last cell of the chain that contains the cell x . In the example of Fig. 3, $Dref(x, r)$ will cause $\bar{r} w$.

The process $(x =_u y)(s, f, b)$ performs unification, which first invokes $Dref$ to dereference possible chains in order to get the last cells x_1 and y_1 :

- If x_1 and y_1 are the same cell, then done.
- If x_1 (resp. y_1) is an unbound variable, then x_1 (resp. y_1) is bound to the cell y_1 (resp. x_1).
- If x_1 and y_1 are both bound, then check whether they have same tag (i.e. same functor/constant name), if yes, then call agent $=_p$ to unify their arguments.

Basically, the agent $=_p$ sequentially calls the unification agent $=_u$ to unify all pairs of the corresponding arguments. Now, let us look at a simple example, illustrated in Fig. 4, where four unbound variables x, y, u, v are to be unified.

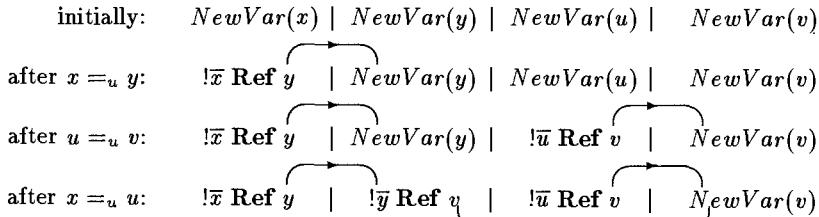


Fig. 4. Unification of Four Variables

In the last step, unification $x =_u u$ is actually performed at cell y and v due to the dereferencing.

4.3 Unification with Backtracking

In Prolog, a backtracked unification must undo all the variable bindings performed during the unification, i.e. reset the corresponding variables to the unbound state. We can modify the above unification agent definitions to incorporate the backtracking, as shown in the Fig. 5. Only the definitions for $BndVar$ and $=_u$ need to be changed. In the bound state $BndVar(x, y)$, a special cell (**Bck**_) can be received along channel x to indicate that x must be reset to the unbound state $NewVar(x)$. So, having reported *success* (s) after binding y_1 to x_1 (resp. x_1 to y_1), the process $(x =_u y)(s, f, b)$ waits on the *backtracking*

$$\begin{aligned}
BndVar(x, y) &\stackrel{\text{def}}{=} \overline{x} \text{ Ref } y.BndVar(x, y) + x(\text{tag } _).[tag = \mathbf{Bck}] NewVar(x) \\
(x=_u y)(s, f, b) &\stackrel{\text{def}}{=} (\nu r_x r_y)(Dref(x, r_x)|Dref(y, r_y)|r_x(x_1).r_y(y_1). \\
&\quad ([x_1 = y_1]\overline{s.b}\overline{.f} + \\
&\quad [x_1 \neq y_1]x_1(\text{tag}_x p_x).y_1(\text{tag}_y p_y). \\
** \rightarrow &\quad ([\text{tag}_x = \mathbf{Var}]\overline{x_1 - y_1.\overline{s.b}\overline{.f}} \mathbf{Bck} \emptyset.\overline{f} + \\
** \rightarrow &\quad [\text{tag}_y = \mathbf{Var}]\overline{y_1 - x_1.\overline{s.b}\overline{.f}} \mathbf{Bck} \emptyset.\overline{f} + \\
&\quad [\text{tag}_x \neq \mathbf{Var}][\text{tag}_y \neq \mathbf{Var}]([\text{tag}_x = \text{tag}_y](p_x =_p p_y)(s, f, b) + \\
&\quad [\text{tag}_x \neq \text{tag}_y]\overline{f}))))
\end{aligned}$$

Fig. 5. Unification with Backtracking

channel b . If it receives a *backtracking* request b , then it will undo the variable binding by sending out a special cell (**Bck**_) along channel $\overline{x_1}$ (resp. y_1) before reporting failure via \overline{f} .

Occur-check is not performed at the above unification, hence circular terms are allowed to be constructed. However, an occur-check can be specified in π -calculus and be easily incorporated into the above unification.

5 Prolog

Now we are going to merge the specifications described in Section 3 and 4 to give a full specification of Prolog. We assume that all Prolog clauses have been simplified in such a way that all head unifications are explicitly called in the body, i.e. all clause heads contain only variables as arguments, such as in the following *append* example:

$$\begin{array}{l} append([], X, X). \\ append([W|X], Y, [W|Z]) :- \quad \quad \quad append(X, Y, Z) : - X = [], Y = Z. \\ \qquad \qquad \qquad append(X, Y, Z). \end{array} \Rightarrow \begin{array}{l} append(X, Y, Z) : - X = [W|X1], Z = [W|Z1], \\ \qquad \qquad \qquad append(X1, Y, Z1). \end{array}$$

5.1 Undo Chains

The agent *Cut* defined in Section 3.3 is not powerful enough when combined with unification. Given a clause in the form: ' $P : -B_1, \dots, B_{j-1}, !, B_{j+1}, \dots, B_n.$ ' a backtracking to the cut $!$ not only causes the failure of the calling process, but also has to undo all the variable bindings performed in B_1, \dots, B_{j-1} . In order to achieve this effect, an *undo* chain is used by associating each goal process with two additional channels of sort **Undo**, i.e. $\llbracket G \rrbracket(u, v)(s, f, b)$ with sorting $\llbracket G \rrbracket : (\text{Undo}, \text{Undo})^{\sim}(\text{Succ}, \text{Fail}, \text{Fail})$. We call v the input *undo* channel, and u the output *undo* channel. Upon receiving an *undo* signal on v , $\llbracket G \rrbracket(u, v)(s, f, b)$ will undo all the variable bindings it has performed, and then send an *undo* signal on \bar{u} , which will cause another process to undo the variable bindings. An *undo* chain runs from right to left in a conjunction, similarly to the *backtracking* chain as shown in Fig. 1. The modified definitions after incorporating the *undo* channels are shown in Fig. 6.

$$\begin{aligned}
(A \triangleright B)(u, v)(s, f, b) &\triangleq (\nu u' s' f')(A(u, u')(s', f, f') \mid !s'.B(u', v)(s, f', b)) \\
(A \triangleright| B)(u, v)(s, f, b) &\triangleq (\nu u' s' f')(A(u, u')(s', f, f') \mid s'.B(u', v)(s, f', b)) \\
(A \oplus B)(u, v)(s, f, b) &\triangleq (\nu f')(A(u, v)(s, f', b) \mid f'.B(u, v)(s, f, b)) \\
Cut(f_0)(u_0)(u, v)(s, f, b) &\stackrel{\text{def}}{=} \bar{s}.(v.\bar{u} + b.([u \neq u_0]\bar{u}.u_0.\bar{f}_0 + [u = u_0].\bar{f}_0)) \\
(x = y)(u, v)(s, f, b) &\stackrel{\text{def}}{=} (\nu s' f' b')((x =_u y)(s', f', b') \mid (f'.\bar{f} + s'.\bar{s}.(b.\bar{b}'.f'.\bar{f} + v.\bar{b}' .f'.\bar{u})))
\end{aligned}$$

Fig. 6. Incorporating the Undo Channels

- Since \triangleright and $\triangleright|$ are used for conjunctions, so their definitions are modified only to expand the *undo* chain, in a similar way as expanding the *backtracking* chain.
- The $Cut : (\text{Fail})^\wedge (\text{Undo})^\wedge (\text{Undo}, \text{Undo})^\wedge (\text{Succ}, \text{Fail}, \text{Fail})$ reports *success* as usual, then
 - if it receives backtracking request b , then it sends \bar{u} to undo all the variable bindings performed by the goals before the cut. As we shall see in the definition (17) in Fig. 7, the u_0 must be the output *undo* channel of the leftmost goal. So signaling on u_0 means that all the undo's have been finished, and only then the *Cut* reports *fail* on \bar{f}_0 . The match $[u = u_0]$ means that the cut itself is the leftmost goal, so no undo is necessary.
 - if it receives *undo* request v , then it passes out the undo request on \bar{u} . This case can happen when the *undo* is caused by other cut or by a *Not*, defined in Section 5.3.
- The unification agent $(x = y) : (\text{Undo}, \text{Undo})^\wedge (\text{Succ}, \text{Fail}, \text{Fail})$ calls the backtracking unification agent $=_u$, defined in Section 4.3. If $=_u$ fails (f'), then it reports *fail* (\bar{f}). If $=_u$ succeeds (s'), then it reports *success* (\bar{s}), but then
 - If a *backtracking* request b is received, then $=_u$ is backtracked (\bar{b}') (which will automatically undo variable bindings as described in Fig. 5). Since a backtracked unification must fail, so after signaling on f' , the agent $=$ reports fails(\bar{f}).
 - If an *undo* request v is received, then by backtracking the $=_u$ via \bar{b}' , all the variable bindings performed by $=_u$ will be undone. After signaling on f' , which also means the undo is finished, the agent $=$ passes the undo request to other goals via \bar{u} .

5.2 Full Specification of Prolog

As in Section 3, suppose that a predicate P of arity k is defined by m clauses, and the i th clause contains a cut $!$ as the j th body goal: ' $P(X_1, \dots, X_k) :- B_1, \dots, B_{j-1}, !, B_{j+1}, \dots, B_n$ '. Let \tilde{x} denotes x_1, \dots, x_k , then a full specification of Prolog is shown in Fig. 7. Compared to the translation in Section 3, there are only the following changes:

- the agents P and P_i now take additional $k+2$ arguments, i.e.
 $P : (\text{Cell}, \dots, \text{Cell})^\wedge (\text{Undo}, \text{Undo})^\wedge (\text{Succ}, \text{Fail}, \text{Fail})$,
 $P_i : (\text{Cell}, \dots, \text{Cell})^\wedge (\text{Fail})^\wedge (\text{Undo}, \text{Undo})^\wedge (\text{Succ}, \text{Fail}, \text{Fail})$.
- *Cut* takes additional argument u' , which is also the output *undo* channel of leftmost process $\llbracket B_1 \rrbracket$, satisfying the requirement stated in Section 5.1.

$$P(\tilde{x})(u, v)(s, f, b) \stackrel{\text{def}}{=} (P_1(\tilde{x}) \oplus \dots \oplus P_i(\tilde{x})(f) \oplus \dots \oplus P_m(\tilde{x}))(u, v)(s, f, b) \quad (16)$$

$$\begin{aligned} P_i(\tilde{x})(f_0)(u, v)(s, f, b) &\stackrel{\text{def}}{=} (\nu v_1 \dots v_s)(\nu u' v') (NewVar(v_1) | \dots | NewVar(v_s) | \\ &(\llbracket B_1 \rrbracket \triangleright \dots \triangleright \llbracket B_{j-1} \rrbracket \triangleright Cut(f_0)(u') \triangleright \llbracket B_{j+1} \rrbracket \triangleright \dots \\ &\triangleright \llbracket B_n \rrbracket) (u', v')(s, f, b) | v. \overline{v'}. u'. \overline{u}) \end{aligned} \quad (17)$$

$$\begin{aligned} \llbracket B(t_1 \dots t_k) \rrbracket(u, v)(s, f, b) &\stackrel{\Delta}{=} (\nu y_1 \dots y_k) (B(y_1, \dots, y_k)(u, v)(s, b, f) | \\ &\llbracket t_1 \rrbracket(y_1) | \dots | \llbracket t_k \rrbracket(y_k)) \end{aligned} \quad (18)$$

with the following restrictions:

- Suppose V_1, \dots, V_s are the all shared local variables among B_1, \dots, B_n , then V_1, \dots, V_s will be translated into $NewVar(v_1) | \dots | NewVar(v_s)$ as in (17).
- in (18), any occurrence of shared local variables (V_1, \dots, V_s) or head variables ($X_1 \dots X_k$) in the body goals will be represented directly by the corresponding $v_1 \dots v_s$ or $x_1 \dots x_k$ without generating new channels.

Fig. 7. Full Specification of Prolog

- $v. \overline{v'}. u'. \overline{u}$ in definition (17) is used to handle the *undo* request (v) from outside of P_i . Hence we distinguish between the *undo* request set by the inside *Cut* which should be stopped at $\llbracket B_1 \rrbracket$, and the *undo* request (v) from outside of P_i which, after finishing all the *undo*'s of this clause (via $\overline{v'}. u'$), should be passed out to other process via \overline{u} .

The restrictions in Fig. 7 require that each shared local variable in the body is translated only once and every occurrence of same variable in different subgoals will use the same channel name (of sort **Cell**). If the i th clause contains no cut, then use $\llbracket B_j \rrbracket$ in place of $Cut(f_0)(u)$ in definition (17) and eliminate the argument (f_0) from P_i . If the i th clause is a fact of the form ' $P(X_1, \dots, X_k)$ ', then the agent P_i is defined as follows:

$$P_i(\tilde{x})(u, v)(s, f, b) \stackrel{\text{def}}{=} \overline{s}.(b. \overline{f} + v. \overline{u})$$

which is also the definition for *true*. As an example, the translation of append is shown in Fig. 8. The $\llbracket \rrbracket$ is the functor name for the list structure.

$$\begin{aligned} Append(x, y, z)(u, v)(s, f, b) &\stackrel{\text{def}}{=} (Append_1(x, y, z) \oplus Append_2(x, y, z))(u, v)(s, f, b) \\ Append_1(x, y, z)(u, v)(s, f, b) &\stackrel{\text{def}}{=} (\nu e)(! \overline{e} \llbracket \emptyset | ((x = e) \triangleright (y = z))(u, v)(s, f, b)) \\ Append_2(x, y, z)(u, v)(s, f, b) &\stackrel{\text{def}}{=} (\nu x_1 z_1 u)(NewVar(x_1) | NewVar(z_1) | NewVar(w) \\ &| (\nu l_1 l_2)((((x = l_1) \triangleright (z = l_2) \triangleright Append(x_1, y, z_1))(u, v)(s, f, b) \\ &| (\nu p_1 p_2)(! \overline{l_1} \llbracket p_1 | ! \overline{p_1} w p_2 | ! \overline{p_2} x_1 \emptyset) \\ &| (\nu q_1 q_2)(! \overline{l_2} \llbracket q_1 | ! \overline{q_1} w q_2 | ! \overline{q_2} z_1 \emptyset))) \end{aligned}$$

Fig. 8. Encoding the *Append* predicate

5.3 Negation as Failure and Other Prolog's Primitives

Based on the principle of *negation as failure*, we can define a control operator *Not* for Prolog negation *not* ($\setminus +$).

$$\begin{aligned} \llbracket \text{not}(P) \rrbracket &\stackrel{\Delta}{=} \text{Not}(\llbracket P \rrbracket) \\ (\text{Not}(A(\tilde{x}))(u, v)(s, f, b)) &\stackrel{\Delta}{=} (\nu u' v' s' f' b') (A(\tilde{x})(u', v')(s', f', b') \mid \\ &\quad (s' \overline{v'} . u' \overline{f} + f' \overline{s} . (b. \overline{f} + v. \overline{u}))) \end{aligned}$$

Agent $\text{Not}(A(\tilde{x}))$ first calls $A(\tilde{x})$: if $A(\tilde{x})$ fails (f'), then $\text{Not}(A(\tilde{x}))$ reports *success* (\overline{s}) and then handles possible backtracking request ($b. \overline{f}$) or passes undo request ($v. \overline{u}$); if $A(\tilde{x})$ succeeds (s'), then $\text{Not}(A(\tilde{x}))$ reports *fail* (\overline{f}) after undoing possible variable bindings (because of $A(\tilde{x})$) via $\overline{v'}. u'$.

Prolog's condition operator ($P \rightarrow Q; R$), as well as *true* and *fail*, are specified in π -calculus as follows:

$$\begin{aligned} \llbracket P \rightarrow Q; R \rrbracket(u, v)(s, f, b) &\stackrel{\Delta}{=} (\nu u' v') (((\llbracket P \rrbracket \triangleright \text{Cut}(f)(u') \triangleright \llbracket Q \rrbracket) \oplus \llbracket R \rrbracket)(u', v')(s, f, b) \\ &\quad | v. \overline{v'}. u'. \overline{u}) \\ \text{True}(u, v)(s, f, b) &\stackrel{\text{def}}{=} \overline{s}. (b. \overline{f} + v. \overline{u}) \\ \text{Fail}(u, v)(s, f, b) &\stackrel{\text{def}}{=} \overline{f} \end{aligned}$$

For the similar reason as in the definition (17) of Fig. 7, $v. \overline{v'}. u'. \overline{u}$ is used in defining the condition agent.

6 Future and Related Works

Ross's CCS Semantics of Prolog The work described here has been more or less influenced by Ross's works on the CCS semantics of Prolog control [RS91, Ros92a] and on the π -calculus semantics of logical variables [Ros92b]. However, our approach, described in Section 3, *exactly* models Prolog's left-right sequential control while [RS91, Ros92a] does not. Namely, given a conjunction goal (P, Q) , its corresponding π -calculus agent $\llbracket P \rrbracket \triangleright \llbracket Q \rrbracket$ works in such a way that the *success* of $\llbracket P \rrbracket$ will invoke $\llbracket Q \rrbracket$, and at the same time the agent $\llbracket P \rrbracket$ will suspend until a backtracking request is received from $\llbracket Q \rrbracket$. However, the corresponding CCS agent $\llbracket P \rrbracket \triangleright \llbracket Q \rrbracket$ in [RS91, Ros92a] works in a different way that the *success* of $\llbracket P \rrbracket$ will invoke $\llbracket Q \rrbracket$, but at the same time the agent $\llbracket P \rrbracket$ can concurrently work to find alternative solutions to invoke another copy of $\llbracket Q \rrbracket$. Hence a certain kind of *or-parallelism* exists, so it is unclear how unification can be incorporated (at the process level) into such a scheme since *or-parallelism* usually requires multi-environments.

The approach of this paper is also simpler since only two³ control operators are needed, corresponding to Prolog's sequential-and and sequential-or, while [Ros92a] uses five control operators. This paper specifies not only Prolog's control but also unification, while [Ros92a] does not specify unification as process. Although a π -calculus semantics of unification has been attempted in [Ros92b], it fails to work for some examples, such as the example of unification of four variables in Fig. 4.

³ \triangleright is only an optimization of \triangleright in the sense that \triangleright works well in the place of \triangleright , as described in Section 3.

Warren's Abstract Machine The representation of logical variables and the approach for unification described in Section 4 are similar to those in the WAM [War83, AK91], especially in the way of using reference chains for binding variables. However, these two approaches are at different levels, i.e. algebraic process level and abstract instruction level. Since the WAM is much closer to an actual implementation than the π -calculus specifications, it necessarily has more complicate and explicit notation of environment stack.

Evolving Algebra The comparison of this work with other semantics specifications of Prolog, especially the work in evolving algebra by Börger and Rosenzweig [BR90, BR92], will be interesting.

Continuation-Passing Style The specification presented in this paper is similar in spirit to the continuation-passing style used in functional programming implementation of Prolog [Hay87, EF91]. In $P(s, f, b)$, the s can be treated as the address of the *success* continuation process which is invoked when P sends \bar{s} ; the f can be treated as the address of *failure* continuation process which is invoked when P sends \bar{f} ; and the b is used to pass the address of *failure* continuation to the next process in the conjunction since b is the same channel as the *failure* channel of next process, as discussed in Section 3. Instead of passing the continuation itself as in most continuation-passing styles, only the address of the continuation is passed in our approach.

Concurrent Logic Programming Languages As the π -calculus is a model of concurrent computation, the approaches of this paper can be extended to specify the semantics of the family of concurrent logic programming languages. As a sequel, some results have been achieved in specifying the flat versions of these languages, such as Flat Parlog[CG86] and Flat GHC[Ued86, Sha89], which do not require multi-environment for *committed or-parallelism* and whose unifications are eventual in the sense that no backtracking is required. The *and-parallelism* is modeled by using the π -calculus concurrent composition ($\|$) processes. The *committed or-parallelism* is modeled in two steps: the *or-parallelism* part is modeled by using concurrent composition ($\|$) processes while the *committed non-determinism* part is modeled by the non-determinism of the π -calculus summation (+) process. We are also successful in specifying a deadlock-free concurrent unification in the π -calculus by associating each logical variable with a lock.

High-Order Features Using the techniques described in [Mil91], we can replace all π -calculus agent definitions with replications. Hence, a predicate definition is now translated into a persistent process which can be called via a unique channel name (served as the address of the process) associated with it. Since a channel name can be passed around, we might be able to extend this technique to specify high-order logic programming languages [Mil90a] in π -calculus. Some preliminary results show this is promising.

7 Conclusion

This paper presents a concise π -calculus specification of Prolog that combines a continuation-passing style specification of control with a WAM style specification of logical variables and unification. Several examples have been tested successfully using a π -calculus interpreter written by the author. Given a π -calculus specification of Prolog, the π -calculus bisimulation theory [Mil89, MPW92b] may be a promising tool in Prolog program transformation and reasoning, following lines developed by Ross in [Ros92a, RS91]. Since the π -calculus is a low level calculus with a simple computation mechanism, it is easy to imagine that a π -calculus specification of Prolog may yield an actual implementation of Prolog.

Acknowledgments: I own many thanks to Dale Miller who directed me to this project and contributed a lot of important ideas presented in this paper. I also thank Srinivas Bangalore for helpful discussions. This project has been funded in part by NSF grants CCR-91-02753 and CCR-92-09224.

References

- [AK91] H. Ait-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, 1991.
- [BR90] Egon Börger and Dean Rosenzweig. From prolog algebras towards wam – a mathematical study of implementations. In *Computer Science Logic (LNCS 533)*, pages 31–66. Springer-Verlag, 1990.
- [BR92] E. Börger and D. Rosenzweig. Wam algebras– a mathematical study of implementation part 2. In *Logic Programming (LNCS 592)*, pages 35–54. Springer-Verlag, 1992.
- [CG86] K.L. Clark and S. Gregory. Parlog: Parallel programming in logic. *ACM Trans. Prog. Lang. Syst.*, 8(1):1–49, January 1986.
- [EF91] C. Elliott and Pfenning F. A semi-functional implementation of a higher-order logic programming language. In P. Lee, editor, *Topics in Advanced Language Implementation*, pages 287–325. The MIT Press, 1991.
- [Hay87] C.T. Haynes. Logic continuations. *Journal of Logic Programming*, 4(2):157–176, 1987.
- [Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
- [Mil90a] D. Miller. Abstractions in logic programming. In P. Odifreddi, editor, *Logic and Computer Science*, pages 329–359. Academic Press, 1990.
- [Mil90b] R. Milner. Function as processes. Technical Report 1154, INRIA, Sophia Antipolis, February 1990.
- [Mil91] R. Milner. The polyadic π -calculus: A tutorial. Technical Report ECS-LFCS-91-180, LFCS, University of Edinburgh, 1991.
- [MPW92a] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, i. *Information and Computation*, 100(1):1–40, 1992.
- [MPW92b] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, ii. *Information and Computation*, 100(1):41–77, 1992.
- [Ros92a] B.J. Ross. *An Algebraic Semantics of Prolog Control*. PhD thesis, University of Edinburgh, Scotland, 1992.
- [Ros92b] B.J. Ross. A π -calculus semantics of logical variables and unification. In *Proc. of North American Process Algebra Workshop*, Stony Brook, NY, 1992.
- [RS91] B.J. Ross and A. Smaill. An algebraic semantics of prolog program termination. In *Eighth International Logic Programming Conference*, pages 316 – 330, Paris, France, June 1991. MIT Press.
- [Sha89] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412–510, September 1989.
- [Ued86] K. Ueda. Guarded horn clause. In *Logic Programming'85*, pages 168–179. LNCS 221, Springer-Verlag, 1986.
- [Wal90] D. Walker. π -calculus semantics of object-oriented programming languages. Technical Report ECS-LFCS-90-122, LFCS, University of Edinburgh, 1990.
- [War83] D.H.D Wareen. An abstract prolog instruction set. Technical Report Note 309, SRI International, Menlo Park, CA, 1983.

A Logical Framework for Evolution of Specifications

Wei Li

Department of Computer Science
Beijing University of Aeronautics and Astronautics
100083 Beijing, P.R. China

Abstract

A logical framework of software evolution is built. The concepts of sequence of specifications and the limit of a sequence are established. Some concepts used in the development of specifications, such as new laws, user's rejections, and reconstructions of a specification are defined; the related theorems are proved. A procedure is given using transition systems. It generates sequences of specifications from a given user's model and an initial specification. It is proved that all sequences produced by the procedure are convergent, and their limit is the truth of the model. Some computational aspects of reconstructions are studied; an R-calculus is given to deduce a reconstruction when a specification meets a rejection. An editor called Specreviser is introduced. It is used to develop specifications. The main functions of the editor are given; some techniques used in its implementation are also discussed. Finally, the theory is compared with AGM's theory of belief revision.

1 Introduction

If we observe the history of development of a software system, we will find that the history can be described by a sequence of versions of the software system:

$$V_1, V_2, \dots, V_n, \dots,$$

where the version V_{n+1} is obtained from V_n either by adding some new pieces of programs, or by correcting some errors, which is done by replacing some pieces of programs of V_n with some new pieces of programs.

Similarly, the history of specifications of a problem can be described by a sequence of drafts of specifications:

$$S_1, S_2, \dots, S_n, \dots,$$

where the draft S_{n+1} is obtained from S_n in the following way: Either clients provide some laws, or we ask some questions (propose some laws); the clients may answer the question by "Yes" or "No". The answer "Yes" means that our proposed laws are accepted and will be added into S_n . The answer "No" means that the laws are rejected by the clients; in this case we say that the law has been rejected by facts.

To build a logical framework, let us consider the specification of a software. From the above discussion, we reach the following conclusions:

1. The history of the development of the specification of a program can be described by the sequence of versions of the specification.

2. Each specification in the sequence should be consistent; otherwise, it fails because anything could be deduced, or equivalently, no program can be synthesized from the specification.
3. When clients reject a specification, the specification has to be revised to meet the client's requirement.
4. There should be a procedure to produce the revisions of the specification in an economical way; i.e., the modification of the specification should be as less as possible, and the sequence made up by the revisions should reach an appropriate specification as fast as possible.

The purpose of this paper is to provide a logical framework for describing the history of development of the specification of a program. In section 2, we will introduce the concept of sequence of formal theories to describe the evolution of a specification, and will further introduce a concept called the limit of sequence to model the result of the evolution of a specification. In section 3, we will introduce some concepts to describe the interaction between the specifications and the users. In section 4, we will define a procedure to generate developing sequences. The procedure is defined using a transition system. We will prove that for a given user's model and a specification, all developing sequences generated from the procedure and starting with the given specification are convergent, and their limit is the laws (the set of truth) of the model. In section 5, we provide another procedure to produce reconstructions when a specification is rejected by the user. In section 6, we will give an introduction to an editor called Specreviser which is built based on the logical framework. Finally, we will compare our work with belief revision theory as given in [AGM 85].

2 Sequences and limits

We assume that the formal language which we use is a first order language L defined in [Gall 87]. We use A , Γ and $Th(\Gamma)$ to denote a formula, a sequence of formulas, and the set of all theorems deduced from Γ respectively.

A sequent is of the form $\Gamma \vdash A$. We employ the proof rules of sequent calculus given in [Paul 87]. Thus, we will treat a sequence of formulas as a set of formulas when it is needed.

A model M is a pair $< M, I >$, where M is a domain and I is an interpretation. Sometimes, we use M_φ to denote a model of a specific problem φ , and use T_{M_φ} to denote the set of all true sentences of M_φ . It is obvious that T_{M_φ} is countable. We use $M \models \Gamma$ to denote $M \models A$ for all A contained in Γ .

The concepts of validity, satisfiability, falsifiability, provability and consistency used in this paper are defined in [Gall 87]. As we know, the proof rules are **sound and complete**.

Definition 2.1 Sequence of specifications

A finite or infinite consistent set (sequence) Γ of closed formulas is called a *specification*. The sentences contained in Γ are called *laws*.

$\Gamma_1, \Gamma_2, \dots, \Gamma_n, \dots$ is called a *sequence of specifications*, or *sequence* for short, if for any n , Γ_n is a specification.

A sequence is increasing (or decreasing) if $\Gamma_n \subseteq \Gamma_{n+1}$ (or $\Gamma_n \supseteq \Gamma_{n+1}$) for all n ; otherwise it is non-monotonic.

In terms of mathematical logic, a specification is in fact a sequence of non-logical axioms, or closed formal theories.

We assume that two sentences P and Q are the same sentence iff $P \equiv Q$ (that is $(P \supset Q) \wedge (Q \supset P)$ is a tautology).

Definition 2.2 Limit of sequence

Let $\{\Gamma_n\}$ be a sequence of specifications. The set of closed formulas:

$$\Gamma^* \equiv \bigcap_{n=1}^{\infty} \bigcup_{m=n}^{\infty} \Gamma_m$$

is called the *upper limit* of the sequence $\{\Gamma_n\}$. The set of closed formulas:

$$\Gamma_* \equiv \bigcup_{n=1}^{\infty} \bigcap_{m=n}^{\infty} \Gamma_m$$

is called the *lower limit* of the sequence $\{\Gamma_n\}$.

A sequence of specifications is *convergent* iff $\Gamma_* = \Gamma^*$. The limit of a convergent sequence is denoted by $\lim_n \Gamma_n$ and is its lower limit (and also the upper limit).

The meaning of the definition above can be seen from the following theorem:

Lemma 2.1 1. $A \in \Gamma^*$ iff there exist infinitely many k_n such that $A \in \Gamma_{k_n}$.
 2. $A \in \Gamma_*$ iff there exists an N such that $A \in \Gamma_m$ for $m > N$.

Proof. Straightforward from the definition. \square

Theorem 2.1 If the sequence $\{\Gamma_n\}$ is increasing (or decreasing), then it is convergent and the limit is $\bigcup_{n=1}^{\infty} \Gamma_n$ (or $\bigcap_{n=1}^{\infty} \Gamma_n$).

Example 2.1 Increasing sequence

We have seen many increasing sequences in logic. For example, consider the Lindenbaum theorem: "Every formal theory Γ of L can be extended to a maximal theory." The proof of the theorem is given as follows: Since all sentences of L are countable, they can be listed as: $A_1, A_2, \dots, A_n, \dots$. We then define $\Gamma_0 = \Gamma$,

$$\Gamma_{n+1} = \begin{cases} \Gamma_n \cup \{A_n\} & \text{if } \Gamma_n \text{ and } A_n \text{ are consistent} \\ \Gamma_n & \text{otherwise} \end{cases}$$

It is obvious that the sequence $\{\Gamma_n\}$ is increasing. Its limit $\bigcup_{n=0}^{\infty} \Gamma_n$ is a maximal theory.

Example 2.2 Sequence without limit

$$\Gamma_n = \begin{cases} \{A\} & n = 2k - 1 \\ \{\neg A\} & n = 2k \end{cases}$$

Thus, $\Gamma^* = \{A, \neg A\}$ and $\Gamma_* = \emptyset$. The sequence $\{\Gamma_n\}$ has no limit.

Example 2.3 Random sequence

Let A denote the statement “*tossing a coin and getting tails.*” Γ_n is defined by the result of n^{th} tossing. The sequence $\{\Gamma_n\}$ is a random sequence of A and $\neg A$. Obviously, it has no limit.

Intuitively, such sequence means that the laws contained in the specification perhaps are not appropriate descriptions of the given problem. For example, an appropriate description of the above example should be “*tossing a coin, the probability of getting tails is 50%.*”

The main result about the limits is the following: There exists a procedure which, for a given model and a given specification which may be inconsistent with the truth of the model, will produce sequences such that every sequence will start with the given specification, and will be convergent to the same limit which is the set of all laws (the truth) of the model.

3 New law and user's rejection

To build the procedure mentioned above, we need the following concepts:

Definition 3.1 New laws.

A is called a new law for Γ iff there exist two models \mathbf{M} and \mathbf{M}' such that

$$\mathbf{M} \models \Gamma, \quad \mathbf{M} \models A \quad \text{and} \quad \mathbf{M}' \models \Gamma, \quad \mathbf{M}' \models \neg A.$$

Theorem 3.1 A is a new law for Γ iff A is logically independent of Γ , that is neither $\Gamma \vdash A$ nor $\Gamma \vdash \neg A$ is provable.

Proof: The proof is straightforward from soundness and completeness. \square

We use $\Gamma \models A$ to denote that A is a semantic consequence of Γ . It means that for any model \mathbf{M} , if $\mathbf{M} \models \Gamma$ then $\mathbf{M} \models A$.

The concept of user's rejection is given below:

Definition 3.2 User's rejection

Let $\Gamma \models A$. A model \mathbf{M} is a user's rejection of A iff $\mathbf{M} \models \neg A$.

Let $\Gamma_{M(A)} \equiv \{A_i \mid A_i \in \Gamma, \quad \mathbf{M} \models A_i, \quad \mathbf{M} \models \neg A\}$.

\mathbf{M} is called an *ideal* user's rejection of A iff $\Gamma_{M(A)}$ is *maximal* in the sense that there does not exist another user's rejection of A , \mathbf{M}' , such that $\Gamma_{M(A)} \subset \Gamma_{M'(A)}$. An ideal user's rejection of A is denoted by $\overline{\mathbf{M}}(A)$.

Since there may exist many ideal rejections of A by facts, we define

$$\mathcal{R}(\Gamma, A) \equiv \{\Gamma_{\overline{\mathbf{M}}(A)} \mid \overline{\mathbf{M}} \text{ is an ideal user's rejection of } A\}$$

The user's rejection meets the intuition that whether a specification is acceptable, depends only on whether all its deduced results agree with user's requirements which have nothing to do with the logical inference. In fact, this is the reason that sometimes, we call our theory *open logic* [Li 92].

The ideal user's rejection satisfies the Occam's razor, which says: “*Entities are not to be multiplied beyond necessity.*” Here, it means that if some particular consequence deduced from a specification is rejected, then only the smallest set of laws (contained in the specification) which cause the rejection has to be rectified, and the rest of the laws (a maximal subset) is retained and is assumed to be temporarily correct.

Definition 3.3 Acceptable modification

Let $\Gamma \vdash A$. An acceptable modification Λ of Γ by $\neg A$ is a maximal subset of Γ with the property that Λ is consistent with $\neg A$.

Let $\mathcal{A}(\Gamma, A)$ be the set of all acceptable modifications of $\neg A$.

Theorem 3.2 $\mathcal{A}(\Gamma, A) = \mathcal{R}(\Gamma, A)$.

Proof: Let us prove \Rightarrow .

Suppose $\Lambda \in \mathcal{A}(\Gamma, A)$. It is consistent with $\neg A$, so there is a model M' such that $M' \models \Lambda$ and $M' \models \neg A$. Thus, M' is a rejection of A by facts. M' is maximal, since if there exists another M'' such that $M'' \models \neg A$ and $\Gamma_{M''(A)} \supset \Gamma_{M'(A)}$, then $\Gamma_{M''(A)} \in \mathcal{A}(\Gamma, A)$; but this is impossible. \square

Example 3.1 Let $\Gamma \equiv \{A, A \supset B, B \supset C, E \supset F\}$

We have $\Gamma \vdash C$. The $\mathcal{A}(\Gamma, C)$ consists of:

$$\{A, A \supset B, E \supset F\}, \quad \{A, B \supset C, E \supset F\} \quad \{A \supset B, B \supset C, E \supset F\}.$$

The following definition tells us how to reconstruct a specification, when we meet a new law or a user's rejection.

Definition 3.4 Reconstruction

Let A be a theorem of Γ . An E-reconstruction of Γ for the theorem A is Γ itself.

Let A be a new law for Γ . An N-reconstruction of Γ for the new law A is the sequence $\{\Gamma, A\}$.

Let $\Gamma \models A$ and A be rejected by the user. An R-reconstruction of Γ for the user's rejection of A is Δ , where $\Delta \in \mathcal{R}(\Gamma, A)$.

Γ' is a reconstruction of Γ iff Γ' is an E- or an N- or an R-reconstruction.

It is obvious that R-reconstruction is not unique. If Δ is an R-reconstruction of Γ for a user's rejection of A , then Δ is a maximal subset of Γ and is consistent with $\neg A$.

The N-reconstruction and R-reconstruction are similar to the expansion and maxichoice contraction in [AGM 85] respectively. The minor difference is that all the concepts in AGM theory are proof-theoretic and are defined for the logical closure $Th(\Gamma)$ (called *belief sets* in [AGM 85]). In contrast, the concepts given here are model-theoretic, and defined for a specification (formal theory) Γ . The key difference is that we are interested in how to build the convergent sequences.

4 The limit of developing processes

Having given the general concepts, we study the problem of how to describe the evolution of specification of a program.

Definition 4.1 Developing process

A sequence of specifications $\Gamma_1, \Gamma_2, \dots, \Gamma_n, \dots$ is a developing process, if Γ_{i+1} is a reconstruction of Γ_i for $i \geq 1$.

It should be mentioned that if $\mathcal{A}(\Gamma_n, A)$ contains more than one element, then there are many R-reconstructions of Γ_n . Thus, the evolution of a specification should be represented by a tree, each branch of which is a developing process.

Theorem 4.1 1. A developing process $\{\Gamma_n\}$ is increasing (or decreasing) iff for all $n \geq 1$, Γ_{n+1} is an N-reconstruction (or R-reconstruction) of Γ_n .

2. A developing process is non-monotonic iff N- and R-reconstructions occur alternatively.

Proof: Straightforward from the definition. \square

Let us now give the procedure mentioned above. We assume that

1. \mathcal{T}_0 is a given countable consistent set of sentences which we accept, and is denoted by $\{A_m\}$.
2. Γ is a given specification. It may be inconsistent with \mathcal{T}_0 , and is to be used as the initial specification of the developing process.

The basic idea of building the procedure is: Take $\Gamma_1 = \Gamma$. Γ_{n+1} is constructed recursively as follows: If $\Gamma_n \vdash A_i$, then take $\Gamma_{n+1} = \Gamma_n$; if A_i is a new law for Γ_n , then Γ_{n+1} is the N-reconstruction of Γ_n for A_i , i.e., $\{A_i; \Gamma_n\}$; if $\Gamma_n \vdash \neg A_i$, that is, $\neg A_i$ has met a user's rejection (A_i is to be accepted), then Γ_{n+1} is an R-reconstruction of Γ_n .

When an R-reconstruction is taken in response to a user's rejection in the n^{th} stage of the developing process, there are two things which we should notice:

1. Only those R-reconstructions containing all new laws accepted before the n^{th} stage are interesting. We introduce a sequence Δ to store the accepted new laws for the steps concerning N-reconstructions.
2. Some information may be lost. For example, consider $\Gamma = \{A \wedge B\}$, both $\Gamma \vdash A$ and $\Gamma \vdash B$ are provable. Assume that A has met a user's rejection, then the maximal subset of Γ which is consistent with $\neg A$ is the empty set. Thus, after the R-reconstruction (of Γ for A), B is missing! In order to repair the loss, we introduce a sequence Θ to collect those A_m for the steps concerning E-reconstructions. After each R-reconstruction, the procedure checks all A_m contained in Θ , and picks up the lost ones back as new laws. Since, at any developing stage j , Θ is always finite, the checking will be terminated.

In order to make the notation easy to understand, we describe the procedure using a transition system [Plo 82] and [Li 82]. We introduce the quadruple:

$$< \mathcal{T}, \Gamma, \Theta, \Delta >$$

to denote the configuration. $head(\mathcal{T})$ and $tail(\mathcal{T})$ are defined as usual. We introduce the operator $*$ to denote the concatenation of a finite sequence with another (finite or infinite) sequence:

$$(A_1, \dots, A_n) * (B_1, B_2, \dots, B_n, \dots) \equiv (A_1, \dots, A_n, B_1, B_2, \dots, B_n, \dots)$$

Definition 4.2 Procedure

Let the initial state of the configuration be

$$< \mathcal{T}_0, \Gamma, \emptyset, \emptyset > .$$

Let $\Gamma_1 = \Gamma$. Γ_{n+1} is defined by the following three rules recursively:

$$\frac{\Gamma \vdash \text{head}(\mathcal{T})}{\langle \mathcal{T}, \Gamma, \Theta, \Delta \rangle \longrightarrow \langle \text{tail}(\mathcal{T}), \Gamma, \Theta * \{\text{head}(\mathcal{T})\}, \Delta \rangle} \quad (1)$$

$$\frac{\Gamma \not\vdash \text{head}(\mathcal{T}) \quad \Gamma \not\vdash \neg\text{head}(\mathcal{T})}{\langle \mathcal{T}, \Gamma, \Theta, \Delta \rangle \longrightarrow \langle \text{tail}(\mathcal{T}), \text{head}(\mathcal{T}) * \Gamma, \Theta, \Delta * \{\text{head}(\mathcal{T})\} \rangle} \quad (2)$$

$$\frac{\Gamma \vdash \neg\text{head}(\mathcal{T}) \quad \Delta \subset \Lambda \quad \Lambda \in \mathcal{A}(\Gamma, \neg\text{head}(\mathcal{T}))}{\langle \mathcal{T}, \Lambda, \Theta, \Delta \rangle \longrightarrow \langle \{\text{head}(\mathcal{T})\} * \Theta * \text{tail}(\mathcal{T}), \Lambda, \Theta, \Delta \rangle} \quad (3)$$

The sequence $\{\Gamma_n\}$ is called a developing process of \mathcal{T}_0 and Γ if it is generated by the above procedure.

Theorem 4.2 Let \mathbf{M}_ρ be a given model, and let Γ be a specification. Every developing process of \mathcal{T}_{M_ρ} and Γ denoted by $\{\Gamma_n\}$ is convergent, and

$$\lim_{n \rightarrow \infty} Th(\Gamma_n) = \mathcal{T}_{M_\rho}.$$

Proof: Let us prove the case that Γ does not contain logically independent laws of \mathcal{T} . Techniques similar to those given in the following proof can be used to prove the theorem without this restriction.

Let $\{\Gamma_n\}$ be a developing process of \mathbf{M}_ρ and Γ . We prove $\lim_n Th(\Gamma_n) = \mathcal{T}_{M_\rho}$ in the following two steps:

1. $\mathcal{T}_{M_\rho} \subseteq (Th(\Gamma))_*$. For any $A_i \in \mathcal{T}_{M_\rho}$, by the construction of $\{\Gamma_n\}$, there must exist an n such that either $A_i \in Th(\Gamma_n)$ or $A_i \notin Th(\Gamma_n)$ and $A_i \in \Gamma_{n+1}$.

- (a) For the first case, by definition 4.2, there must be an l such that $A_i \in Th(\Gamma_m)$ for $m \geq l$, since \mathcal{T}_{M_ρ} is consistent. For each $m \geq l$, there is a finite subset of Γ_m denoted by $\Delta_m = \{B_{m,1}, \dots, B_{m,j}\}$ and $\Delta_m \vdash A_i$.

For each k , $1 \leq k \leq j$, either $B_{m,k} \in \bigcap_{n=l}^{\infty} \Gamma_n$ or $(\bigcap_{n=l}^{\infty} \Gamma_n) \vdash B_{m,k}$, otherwise by definition 4.2, $B_{m,k} \in \Gamma$, and there must exist an $m' \geq m$ such that $\Gamma_{m'}$ contains $\neg B_{m,k}$, thus $B_{m,k} \notin Th(\Gamma_{m'})$ which is a contradiction. Thus $A_i \in \bigcap_{m=l}^{\infty} Th(\Gamma_m)$ holds. Therefore

$$A_i \in \bigcup_{n=1}^{\infty} \bigcap_{m=n}^{\infty} Th(\Gamma_m) = (Th(\Gamma))_*.$$

- (b) For the second case, by the definition 4.4, $A_i \in \Gamma_m$ for any $m > n$. Thus,

$$A_i \in \bigcup_{n=1}^{\infty} \bigcap_{m=n}^{\infty} \Gamma_m = \Gamma_*.$$

2. $Th(\Gamma)^* \subseteq \mathcal{T}_{M_\rho}$. Assume that there is a closed formula A such that $A \in Th(\Gamma)^*$ and $A \notin \mathcal{T}_{M_\rho}$. There are only two possibilities:

- (a) neither $\mathcal{T}_{M_p} \vdash A$ nor $\mathcal{T}_{M_p} \vdash \neg A$ is provable, but this contradicts that Γ does not contain any logically independent formula w.r.t. \mathcal{T}_{M_p} .
- (b) $\neg A \in \mathcal{T}_{M_p}$. This is also impossible, if so, there must be i such that $\neg A = A_i$, then there must be n such that $\neg A \in \Gamma_n$. Thus $\neg A \in \Gamma_m$ for all $m > n$. this is $\neg A \in \Gamma^*$, a contradiction.

Thus, we have

$$\text{Th}(\Gamma)^* \subseteq \mathcal{T}_{M_p} \subseteq \text{Th}(\Gamma)_*$$

and so $\text{Th}(\Gamma)_* = \text{Th}(\Gamma)^*$. Hence, the theorem has been proved. \square

The \mathcal{T}_{M_p} can be viewed as the set of laws characterizing a specific problem M_p in the real world; then the above theorem says that we can start from any given specification (which may be inconsistent with M_p) and produce versions of specifications using the above procedure; all these sequences generated are convergent to the set of laws characterizing the problem.

The procedure also shows that when we deal with a user's rejection of A at any stage n , we can arbitrarily choose one maximal set of Γ_n which is consistent with $\neg A$. If it contains Δ which is the set of new laws $A_k \in \mathcal{T}_{M_p}$ accepted in the 1st to n^{th} stage, then the developing process will always converge to \mathcal{T}_{M_p} . Therefore, we do not need the unique maximal consistent subset (which is required in [Gär 88]).

5 A calculus of R-reconstruction

From the definition given in section 3, we know that the key point of building the reconstructions for a specification Γ is to find the maximal subsets of Γ which are consistent with some given formula A . The following two theorems show that this is not an easy task.

- Theorem 5.1**
1. If a specification Γ contains finite propositions only, then building a reconstruction of Γ is an NP-hard problem.
 2. If a specification Γ is a set of first order formulas, then building a reconstruction of Γ is an undecidable problem.

Proof: The first is directly from Cook's theorem [GJ 79]; the second is from Gödel's second incompleteness theorem [Shoen 67]. \square

More detailed results about complexity for propositional logic can be found from [EG 92].

According to the theorem in the last section, when $\Gamma \vdash A$ and A has met a user's rejection, any R-reconstruction of Γ for A (containing Δ) can be chosen, and the limit of developing process does not change. In this section we define a calculus to produce one R-reconstruction. It is called R-calculus, and is defined using transformation rules. The purpose of the rules is to delete the sentences in Γ which contradict $\neg A$. We assume that Γ contains finite sentences, and use the form:

$$\Delta | \Gamma$$

to denote a configuration which is read as Δ overrides Γ . In particular, if $\Delta = A$, then it means that either $\Gamma \vdash \neg A$ and $\neg A$ has met a user's rejection (i.e., A has to be accepted), or A and Γ are consistent. We use

$$\Delta \mid \Gamma \implies \Delta' \mid \Gamma'$$

to mean that the configuration $\Delta \mid \Gamma$ is transformed to $\Delta' \mid \Gamma'$. \implies^* is used to denote the transition closure as usual. In particular, the form

$$\Delta \mid A, \Gamma \implies \Delta \mid \Gamma$$

means that $\Delta \mid \Gamma, A$ is transformed to $\Delta \mid \Gamma$, and A is called a deleted formula in the transformation. In this case, A contradicts Δ . The rules of R-calculus are given below:

Definition 5.1 R-calculus

Structural rules

Contraction

$$\Delta \mid A, A, \Gamma \implies \Delta \mid A, \Gamma \quad A, A, \Delta \mid \Gamma \implies A, \Delta \mid \Gamma$$

Exchange

$$\Delta \mid A, B, \Gamma \implies \Delta \mid B, A, \Gamma \quad A, B, \Delta \mid \Gamma \implies B, A, \Delta \mid \Gamma$$

Logical rules

R- \wedge left rule:

$$A \wedge B, \Delta \mid \Gamma \implies A, B, \Delta \mid \Gamma$$

R- \wedge right rule:

$$\frac{\Delta \mid A, \Gamma \implies \Delta \mid \Gamma}{\Delta \mid A \wedge B, \Gamma \implies \Delta \mid \Gamma} \quad \frac{\Delta \mid B, \Gamma \implies \Delta \mid \Gamma}{\Delta \mid A \wedge B, \Gamma \implies \Delta \mid \Gamma}$$

R- \vee left rule:

$$A \vee B, \Delta \mid \Gamma \implies A, \Delta \mid \Gamma \quad A \vee B, \Delta \mid \Gamma \implies B, \Delta \mid \Gamma$$

R- \vee right rule:

$$\frac{\Delta \mid A, \Gamma \implies \Delta \mid \Gamma \quad \Delta \mid B, \Gamma \implies \Delta \mid \Gamma}{\Delta \mid A \vee B, \Gamma \implies \Delta \mid \Gamma}$$

R- \supset left rule:

$$A \supset B, \Delta \mid \Gamma \implies \neg A, \Delta \mid \Gamma \quad A \supset B, \Delta \mid \Gamma \implies B, \Delta \mid \Gamma$$

R- \supset right rule:

$$\frac{\Delta \mid \neg A, \Gamma \implies \Delta \mid \Gamma \quad \Delta \mid B, \Gamma \implies \Gamma}{\Delta \mid A \supset B, \Gamma \implies \Delta \mid \Gamma}$$

R- \neg left rule:

$$\neg A, \Delta \mid A, \Gamma \implies \neg A, \Delta \mid \Gamma$$

R- \neg right rule:

$$A, \Delta \mid \neg A, \Gamma \implies A, \Delta \mid \Gamma$$

R- \forall left rule:

$$\forall x. A, \Delta \mid \Gamma \implies A[y/x], \forall x. A, \Delta \mid \Gamma$$

R- \forall right rule:

$$\frac{\Delta \mid A[t/x], \Gamma \implies \Delta \mid \Gamma}{\Delta \mid \forall x. A, \Gamma \implies \Delta \mid \Gamma}$$

R- \exists left rule:

$$\exists x. A, \Delta \mid \Gamma \implies A[t/x], \exists x. A, \Delta \mid \Gamma$$

R- \exists right rule:

$$\frac{\Delta \mid A[y/x], \Gamma \implies \Delta \mid \Gamma}{\Delta \mid \exists x. A, \Gamma \implies \Delta \mid \Gamma}$$

In the quantifier rules, x is any variable and y is any variable free for x in A and not free in A unless $y = x$ ($y \notin FV(A) - \{x\}$). The term t is any term free for x in A . In both R- \forall left and R- \exists right rules, the variables y does not occur free in the lower sequent.

R- \neg substitution rules:

$$A, \Delta \mid \Gamma \implies A', \Delta \mid \Gamma \quad \Delta \mid A', \Gamma \implies \Delta \mid A, \Gamma$$

In the \neg substitution rules, A and A' are defined below:

A	$\neg(B \wedge C)$	$\neg(B \vee C)$	$\neg\neg B$	$\neg(B \supset C)$	$\neg\forall x. B$	$\neg\exists x. B$
A'	$\neg B \vee \neg C$	$\neg B \wedge \neg C$	B	$B \wedge \neg C$	$\exists x. \neg B$	$\forall x. \neg B$

R- Γ consistency rule:

$$\frac{\Gamma \vdash \neg A}{\Delta \mid A, \Gamma \implies \Delta \mid \Gamma}.$$

$\Delta \mid \Gamma$ is called a terminated configuration if no one of the above rules can be applied to.

Informally, the right rules are used to delete those formulas which occur in the right hand side of a configuration and contains a formula occurring in the left side of the configuration as a component; the left rules are used to decompose the formulas occurring in the left hand side of a configuration. For the R-calculus, we can prove the following theorem:

Theorem 5.2 If $A \mid \Gamma \Rightarrow^* \Delta' \mid \Gamma'$ and $\Delta' \mid \Gamma'$ is a terminated configuration, then Γ' is an R-reconstruction of Γ for A .

Proof: The theorem can be proved using the techniques given in chapter 5 of [Gal 87], since the theorem is, in fact, talking about a kind of completeness. We omit the proof since it is too long.

Let us study some examples to see how to use the rules.

Example 5.1 Consider the example given in section 3. Let

$$\Gamma \equiv \{A, A \supset B, B \supset C, E \supset F\} \text{ and } \Gamma' \equiv \{A, A \supset B, E \supset F\}.$$

$\Gamma \vdash C$ is provable. Assume that C has met a user's rejection. According to the definition, Γ' is an R-reconstruction of Γ . Using the R-calculus we have:

$$\frac{\neg C \mid \neg B, \Gamma' \Rightarrow^3 \neg C \mid \Gamma' \quad \neg C \mid C, \Gamma' \Rightarrow^2 \neg C \mid \Gamma'}{\neg C \mid B \supset C, \Gamma' \Rightarrow^1 \neg C \mid \Gamma'}.$$

where \Rightarrow^1 is by R- \supset right rule; \Rightarrow^2 is by the Axiom; and \Rightarrow^3 is by R- Γ consistency since $\Gamma' \vdash \neg\neg B$.

Example 5.2 Let $\Gamma \equiv \forall x.A(x), \Gamma'$. It is proved that

$$\Gamma \vdash \neg\exists x.\neg A(x).$$

Assume that $\neg\exists x.\neg A(x)$ has been rejected, i.e., We have to accept $\exists x.\neg A(x)$. Using R-calculus we have:

$$\exists x.\neg A(x) \mid \forall x.A(x), \Gamma' \Rightarrow^1 \neg A(t/x), \exists x.\neg A(x) \mid \forall x.A(x), \Gamma'$$

and

$$\frac{\neg A(t/x), \exists x.\neg A(x) \mid A(t/x), \Gamma' \Rightarrow^3 \neg A(t/x), \exists x.\neg A(x) \mid \Gamma'}{\neg A(t/x), \exists x.\neg A(x) \mid \forall x.A(x), \Gamma' \Rightarrow^2 \neg A(t/x), \exists x.\neg A(x) \mid \Gamma'}$$

Where \Rightarrow^1 is by R- \exists left rule, \Rightarrow^2 by R- \forall right rule, and \Rightarrow^3 by the Axiom.

As we mentioned, using the above R-calculus, only one R-reconstruction can be deduced. For example, we can deduce the R-reconstructions $\{A, A \supset B, E \supset F\}$, but not the other two: $\{A, B \supset C, E \supset F\}$ and $\{A \supset B, B \supset C, E \supset F\}$ given in Example 3.1.

To obtain all R-reconstructions, we believe that a new rule has to be introduced. To do so, we need to define an order:

Definition 5.2 For every right rule of R-calculus, the formula (in the lower transition) to which the rule is applied is called the principle formula, the formulas introduced in the upper transitions are called the side formulas. An order \prec is defined recursively as below:

1. If P is a side formula and Q is the principle formula in an instance of some right rule of R-calculus, then $P \prec Q$.
2. If $P \prec Q$ and $Q \prec R$, then $P \prec R$.

The new rule is defined as:

R-selection rule

$$\frac{\Delta \mid P, Q, \Gamma \Rightarrow \Delta \mid Q, \Gamma \quad Q \prec P}{\Delta \mid P, Q, \Gamma \Rightarrow \Delta \mid P, \Gamma}$$

Having defined the R-selection rule, we have naturally reached the following conjecture:

For a give $A \mid \Gamma$ where $\Gamma \vdash \neg A$ and A has to be accepted, using R-calculus and R-selection rule we can deduce all R-reconstructions.

In fact, the order \prec satisfies the entrenchment relation given by Gärdenfors and Makinson. The R-calculus, the order \prec and the R-selection rule together make the R-reconstructions deducible.

6 The Specreviser

There is a new editor called Specreviser which is built based on the above theory [Li 93]. A prototype of the editor has been implemented on a Sun workstation. To explain its functions, let us study the following example:

Example 6.1 Alarm control

Assume that we have acquired the laws of an Alarm control from the clients. It includes four laws:

1. An Alarm will be raised within 20 seconds, when the device senses an abnormal condition.
2. The alarm will continue as long as it is abnormal.
3. The alarm lasts for at least 3 seconds.
4. The alarm will stop in 5 seconds after the device is recovered.

Our task is to write a specification for these laws. We assume that the first and the fourth items are crucial. We call them **marked laws**. This means that they cannot be changed even if they contradict with some other laws. For simplicity, we use:

$ab(x)$ to denote that at time x the device is abnormal, and
 $raisealarm(x)$ to denote that at time x alarm is raised, and
 $workalarm(x)$ to denote that at time x the alarm is active,
 $(x \leq t \leq x + y)$ to denote $(x \leq t) \wedge (t \leq x + y)$, and $t \in [x, y]$ denotes $x \leq t < y$.

The user's first step in developing a specification might include the following (not necessarily correct) representations of these laws:

1. The first requirement:

$$(1^*). \quad \begin{aligned} \forall x. \exists y. y < x \wedge \forall t. (y \leq t < x \wedge \neg ab(t)) \wedge ab(x) \supset \\ (\exists z. ((x \leq z \leq x + 20) \wedge raisealarm(z))). \end{aligned}$$

This formula says that: 1). At time x the device becomes abnormal. 2). There exists time $y < x$, such that in the period $[y, x)$ the device was normal. 3). Alarm will be raised in the period $[x, x + 20]$. 4). It is a **marked** requirement denoted by a $*$!

2. The second requirement:

$$(2^*). \quad \forall t. workalarm(t) \supset ab(t).$$

This formula means that abnormal status of the device is a necessary condition of working status of the alarm.

3. The third requirement:

$$(3^*). \quad \forall x. raisealarm(x) \supset \forall t. (x \leq t \leq x + 3 \supset workalarm(t))$$

It means that: 1). If at time x the alarm is raised. 2). then in the time interval $[x, x + 3]$ the alarm works. 3). This requirement is **marked** by a $*$.

Having written the third law, we may think that our specification by now is working well. In fact, a careful reader has already found that the specification **fails** because law (3^*) contradicts with law (2) !

The trouble can be seen more clearly through the following counter example: Assume that at time t_1 the device became abnormal; at time $t_2 = t_1 + 5$ the alarm is raised; and then at time $t_3 = t_1 + 6$ the device is recovered. Then, according to the third law, in the time interval $[t_1 + 6, t_1 + 8]$, the alarm is working, meanwhile the device is normal.

Since the third law is **marked**, we have to change the formula representing of the second law. It could be:

$$(2') \quad \begin{aligned} \forall x. ab(x) \wedge (\exists t. x < t \wedge raisealarm(t) \supset \\ \exists z. \forall y. (t < y \leq z \wedge ab(y) \wedge workalarm(y))) \end{aligned}$$

It says that: 1) If at x the device is abnormal and at time $t > x$ the alarm is raised; 2) then there exists $z > t$ such that in the time interval $[t, z]$ the alarm does not stop as long as the device is abnormal; Law $2'$ is consistent with laws 1 and 3.

This example shows that when we build a specification, usually it is very difficult for a user to check whether his specification is consistent, especially when the specification is large — for example, it contains more than 300 laws. In fact, there is a need to build some software tools to **interactively** check the consistency of the specification in every stage of its development.

Our Specreviser is a software system to meet the need. It works like an editor. The new versions of the specification are produced by Specreviser.

Like syntax-directed editors and synthesizers (including type and proof editors), the Specreviser works interactively with the user. It not only points out the syntactic or static errors of the law entered, but also checks the consistency of the newly added laws with respect to the specification which has already been stored.

The Specreviser detects the fallibility of the specification stored when a user's rejection arises. It will point out all the minimal subsets of laws which cause the rejection.

The Specreviser will further provide all possible reconstructions to cope with the user's rejection, and ask the user to select their preferred revision.

We have proved that a user can choose one any of the possible solutions, and the revised specification will eventually reach all the laws characterizing the problem. The difference between an expert and inexperienced user is that the latter may take more steps to reach the goal.

The Specreviser allows the user to mark some laws to mean that they are crucial and cannot be rejected. If a newly added law contradicts some marked laws, then it cannot be added into the specification; if it is also marked, then the Specreviser will point out the contradiction and ask the user to re-mark the laws.

The Specreviser will direct the user in creating an appropriate specification while maintaining the consistency of the specification at each stage of the developing process.

In practice, a user usually may not know the mathematical model of a problem at the beginning of a developing process. However, to build a reconstruction does not require the user to know the whole model of the problem. The famous Chinese Philosopher Confucius claimed a criterion which says that "*truth comes from practice.*" Here it means that a theory must be modified if its logical consequences fail to agree with our observations and experiments; and a statement must be replaced by another if neither itself nor its negation agrees with practice. If one follows this criterion, then he builds his model during development and eventually reaches an appropriate specification by making the reconstructions.

7 Related work

As we have mentioned that the concepts which are similar to the reconstructions were first given in the AGM theory ([AGM 85]), where they call them expansion, contraction and revision. The differences between their approach and ours are the following:

1. AGM focused on setting up a formal theory of revisions of belief sets, i.e., the postulates of revisions. In contrast, our goal is to build a theory of *sequences* of formal theories to model the evolution of knowledge. The belief set is a special specification satisfying $\Gamma = Th(\Gamma)$.
2. AGM introduced the proof-theoretic concepts of expansion, contraction, and revision and studied their properties. In contrast, we use the model-theoretic concepts such as new law and rejection by facts to describe the interactions between logical inference (logical information) and human experience (empirical information). We also set up the relations between the model-theoretic concepts and the proof-theoretic concepts.
3. We introduced the concept of limit of sequences to model the result of the evolution of specifications. We gave a procedure to build developing processes from a given model and a specification. We further proved that the developing processes generated from the procedure are started at the specification and are convergent to the truth of the model.
4. The theorem shows that *at any stage n* of a developing process, we can choose any R-reconstruction (containing Δ) and the process will always have the same limit. Therefore, the unique revision (R-reconstruction) required by AGM is not necessary. A precise description of the uniqueness

of revisions may be expressed as: For a given model and a specification, the limit of all developing processes generated by the procedure is unique.

5. We have introduced the R-calculus to deduce the R-reconstructions when a specification meets a rejection.

Finally, it should be pointed out that the concepts and results given in this paper can be applied to any formal languages with a complete proof system; and provide a framework for knowledge base maintenance, machine learning, software engineering, and diagnostic techniques.

Acknowledgement

I would like to thank Zhang Yuping for helpful discussions. The work is supported by Chinese Science Foundation and Chinese High-Tech Programme.

References

- [AGM 85] Alchourrón, C.E., Gärdenfors, R. and Makinson, D., On the logic of theory change: partial meet contraction and revision functions, *The Journal of Symbolic Logic*, Vol.50, No.2, June, 1985.
- [EG 92] Eiter, T. and Gottlob, G., On the complexity of propositional knowledge base revision, *Artificial Intelligence*, Vol. 57, October, 1992.
- [Gall 87] Gallier, J.H., *Logic for Computer Science, foundations of automatic theorem proving*. John Wiley & Sons, 1987, 147-158, 162-163, 197-217.
- [GJ 79] Garey, M.R. and Johnson, D.S., *Computers and Intractability*, Freeman and Company, 1979.
- [Li 92] Li, W., An Open Logic System, *Scientia Sinica, Series A*, Oct, 1992 in Chinese, March, 1993 in English.
- [Li 93] Li W., A Theory of Requirement Capture and its Applications, TAP-SOFT'93, LNCS 668, 1993, Springer-Verlag.
- [Paul 87] Paulson, L., *Logic and Computations*, Cambridge University Press, 1987, 38-50.
- [Plo 82] Plotkin, G., An structural approach to operational semantics, Lecture notes, Arhus University, Denmark, 1982.
- [Shoen 67] Shoenfield, J.R., *Mathematical Logic*, Addison-Wesley, Reading, Mass, 1967, 74-75.

A Semantics for Higher-order Functors

David B. MacQueen¹ and Mads Tofte²

¹ AT&T Bell Labs, New Jersey, USA

² Dept. of Computer Science, Copenhagen University, Denmark

Abstract. Standard ML has a module system that allows one to define parametric modules, called *functors*. Functors are “first-order,” meaning that functors themselves cannot be passed as parameters or returned as results of functor applications. This paper presents a semantics for a higher-order module system which generalizes the module system of Standard ML. The higher-order functors described here are implemented in the current version of Standard ML of New Jersey and have proved useful in programming practice.

1 Introduction

One of the notable characteristics of the Standard ML module system has been its support of parameterization in the form of *functors*, which are mappings from ordinary modules, called *structures*, to ordinary modules. In the original Standard ML module system ([7, 10]), functors were first-order, because their parameters and results could only be structures, and functors could not be components of structures. But the type theoretic analysis of the module system carried out in [8, 11, 5] made it clear that it was natural to extend the notion of functors to higher orders by allowing functors as parameters and results (or, equivalently, allowing structures to contain functor components). Doing so makes the language more symmetrical and supports useful new modes of parameterization.

A practical implementation of higher-order functors has recently been provided in the Standard ML of New Jersey compiler [3]. The first step toward defining a semantics of higher-order functors was taken in [14], where a semantics for functor signatures is described and a principal signature theorem is proved. Here we go most of the way toward completing the semantics of higher-order functors by defining how functors are represented, how higher-order signature matching is performed, and how functor application works.

The technical challenge in defining a semantics of higher-order functors arises from the way static identity information is propagated in Standard ML. Signature matching is “transparent” by default, meaning that the identities of type and structure components are not hidden when a structure is matched against a signature. Also, identities are propagated through functor calls. This is a controversial feature of the design, but it is justified because (1) it allows a single semantics of signature matching to work both for parameter constraints and result constraints, and (2) it increases the expressiveness and flexibility of parameterization in useful ways.

Alternative module system designs that do not use transparent signature matching have been proposed. For instance, the Extended ML specification language [13], which is based on Standard ML, assumes that signature matching is opaque, and recently Leroy [6] and Harper and Lillibridge [4] have described module systems that use opaque signature

matching but allow one to override it in the case of types by using type definitions in signatures. However, in the higher-order system these proposals produce an asymmetry between first-order and higher-order functors: types in a functor result can depend on structure parameters, but not on functor parameters (see the example below).

We want to avoid this asymmetry between structures and functors, so in our semantics functor parameters as well as structure parameters can carry type information that is propagated to the result.

As an illustration of how application of a first-order functor propagates static identities, consider the following example:

```

signature POINT =
sig
  type point
  val leq: point*point->bool
end;
signature INTERVAL =
sig
  type interval and point
  val mk: point*point -> interval
  val left: interval -> point
  val right: interval -> point
end;
functor Interval(P: POINT): INTERVAL =
struct
  type interval = P.point * P.point
  type point = P.point
  fun mk(x,y) = if P.leq(x,y) then (x,y) else (y,x)
  fun left(x,_) = x
  fun right(_,y) = y
end;
structure IntPoint =
struct
  type point = int; (*1*)
  fun leq(x:int,y) = x<=y
end;
structure T = Interval(IntPoint);
val test = T.right(T.mk(3,4))+5;

```

This program is legal Standard ML. The declaration of `test` is type-correct because the application `Interval(IntPoint)` propagates the information `point = int` (declared at line `(*1*)`) through to `T`, so that `T.interval` is `int*int` and `T.mk` and `T.right` have types `int*int->int*int` and `int*int->int`, respectively. (Notice that if types were *not* propagated through the functor application, the declaration of `test` would be illegal.)

Now let us add a higher-order functor:

```

functor G(functor Interv(P: POINT): INTERVAL) =
struct
  structure NatNumInt = Interv(IntPoint)
end

```

Since the actual functor `Interval` matches the specification in the parameter signature for `G`, it should be possible to apply `G` to `Interval`:

```
structure Result = G(functor Interv = Interval)
structure T' = Result.NatNumInt;
val test' = T'.right(T'.mk(3,4))+5
```

But will the expression `T'.right(T'.mk(3,4))+5` be type-correct? The point is that the parameter signature of `G` did not specify sharing between the argument and the result signature of `Interv`. Thus when the declaration of `G` was elaborated, there was no assumption of sharing between the point type `P.point` and the point type `NatNumInt.point`.

The actual functor, `Interval`, propagates more sharing than is specified for `Interv`. Were we to elaborate the body of `G` again, this time using the actual `Interval` in place of `Interv`, the declaration of `test'` would be legal; if we ignore the extra sharing, however, the declaration of `test'` becomes untypable.

One could argue that this problem is easily solved by making the specification of `Interv` more specific so that it expresses the sharing required:

```
functor G(X:
  sig
    functor Interv(P: POINT):
      sig
        type interval
        val mk: P.point * P.point -> interval
        val left: interval -> P.point
        val right: interval -> P.point
      end
    end)=
  struct
    structure NatNumInt = Interv(IntPoint)
  end
```

But after this change we can only apply `G` to arguments that satisfy this extra sharing, which was not needed inside the body of the functor `G`, so `G` is less general than it could be.

More generally, consider the declaration of some functor, `F`. Is it sufficient to specify the parameter signature of `F` with sharing constraints that are needed to elaborate the body of `F`, or is it necessary to specify any sharing that must be propagated at some application of `F`? From a programmer's point of view, the former is clearly preferable and it is the policy followed in Standard ML. To preserve this desirable property of Standard ML in the presence of higher-order functors, our static semantics of modules must be able to propagate additional type information at functor application time, even when the additional information comes from functor components of the actual argument. So to properly treat a functor application embedded in a functor body, such as `Interv(IntPoint)` in the body of `G`, we must elaborate it in two phases: first formally, when `G` is defined, and then again when `G` is applied and additional sharing information about the actual parameter is available.

In the remainder of this paper we first present the semantic objects (Section 2). Then we give a grammar for a skeletal programming language and elaboration rules

in terms of the semantic objects (Section 3). The key ideas for achieving the desired propagation of type information are (1) using terms in a simple higher-order language to represent functors, and (2) using two environments to simulate the two phase elaboration of embedded functor applications.

The skeletal language we present has neither types nor values, but we foresee no serious problems in extending the semantics to cope with these because the interaction between module systems and the core ML language is well understood. We also do not deal with elaboration of signature expressions in this paper; this was studied in detail in [14].

In addition to the work on Extended ML and the work of Leroy and Harper and Lillibrige already cited, the work of Aponte [1] should also be noted. It provides another approach to semantic representations for ML modules, based on Rémy's work on polymorphic records [12]. So far, this approach deals with first-order functors only.

2 Semantic objects

Our semantic objects are defined informally using a mixture of simple mathematical constructions (*e.g.* sets of sequences of identifiers) and term structures (*e.g.* lambda abstractions) over these constructions. The representations are finite, and in principle they could all be defined uniformly by an abstract syntax of terms.

In the skeletal language, a structure S can have two kinds of named components: structures and functors. The *substructures* of S are S and the substructures of the structure components of S . We say that a functor is (*embedded*) in a structure S , if it is a component of S or of one of the substructures of S . Our representation of structures is based on separating the “shape” of a structure, which defines what is accessible, from the static information that identifies the elements of the structure. The former is represented by a tree s (Section 2.1) of access paths for substructures and embedded functors, and the latter by a realization φ (Section 2.3), which represents a mapping from these paths to identifying information. A *structure* is then defined to be a pair $\langle s, \varphi \rangle$.

2.1 Identifiers, paths, and trees

We assume two disjoint sets of identifiers:

$$\begin{array}{lll} \textit{funid} & \in & \text{FunId} \\ \textit{strid} & \in & \text{StrId} \end{array} \quad \begin{array}{l} \text{(functor identifier)} \\ \text{(structure identifier)} \end{array}$$

Substructures and embedded functors are accessed via paths of identifiers. Formally, a structure path, sp , is a finite string over the alphabet StrId. We also use the notation $strid_1 \dots strid_k$, ($k \geq 0$) for structure paths. A functor path, fp , is a finite string over the alphabet StrId \cup FunId of the form $strid_1 \dots strid_k.funid$, ($k \geq 0$), *i.e.*, a structure path followed by a functor identifier. A path, p , is either a structure path or a functor path. The empty path is denoted ϵ .

A tree, s , is a finite, prefix-closed set of paths. Let s be a tree and assume $p \in s$. Then the subtree of s at p , written s/p , is the tree $\{p' \mid pp' \in s\}$. When s is a tree, $SP(s)$ denotes the set of structure paths in s and $FP(s)$ denotes the set of functor paths in s .

2.2 Stamps

Stamps are used to statically identify structures, and are the basis for determining sharing: two structures share if and only if they are labeled by the same stamp. Only structures have stamps — functors do not have a static identity, though they do have static descriptions.

We assume a denumerably infinite set Stamp of *stamps*. We use m to range over stamps. A *stamp set* is a finite set of stamps. We use M and N to range over stamp sets.

2.3 Realizations

Intuitively, the realization part of a structure is a mapping over the structure's path tree that takes structure paths to stamps and functor paths to static functor representations. However, it turns out to be useful to talk about realization expressions, rather than the maps they denote; realization expressions are defined in Figure 1. Signatures (Σ) will be defined in Section 2.4.

Realization environments and views are concrete representations of finite maps. The domain of a realization environment β , written $\text{Dom}(\beta)$, is defined by: $\text{Dom}(\{\}) = \emptyset$ and $\text{Dom}(\beta', \rho=\varphi) = \{\rho\} \cup \text{Dom}(\beta')$ and similarly for views. We allow repeated binding of the same domain element, with the convention that bindings to the right supersede bindings to the left. We write, for example, $\beta(\rho)$ to denote the realization to which β binds ρ , when $\rho \in \text{Dom}(\beta)$. We often write realization environments out in full, with the notation $\rho_1=\varphi_1, \dots, \rho_n=\varphi_n$ (dropping the initial $\{\}$). Realization environments β_1 and β_2 can be appended, written $\beta_1 + \beta_2$.

Realization Expressions (φ)		Views (η)	
$\varphi ::= (\mu, \eta)$	stamping	$\eta ::= \{\}$	empty
ρ	realization variable	$\eta, strid=\varphi$	extension
$\varphi / strid$	substructure selection	$\eta, funid=\theta$	extension
$\text{app}(\theta, \varphi)$	functor application		
$\varphi \downarrow \Sigma$	signature constraint		
$\text{new } N.\varphi$	generative stamps		
$\text{let } \beta \text{ in } \varphi$	local binding	$\theta ::= \lambda \rho : \Sigma.(s, \varphi)$	functor
		$\text{get}_P(\varphi, fp)$	application
Realization Environments (β)		Stamp Expressions (μ)	
$\beta ::= \{\}$	empty environment		
$\beta, \rho=\varphi$	extension	$\mu ::= m$	stamp
		$\text{get}_S(\varphi, sp)$	application

Fig. 1. Basic semantic representations

To get an idea of the meaning of the constructs of Figure 1, consider the following functor declaration:

```

functor G(X: sig
          functor Interv(P: POINT): INTERVAL
          structure I: POINT
        end)=
  struct
    structure NatNum = X.Interv(X.I)
  end;

```

This declaration gives rise to the view $\eta_G \equiv G = \lambda \rho : \Sigma_X. \langle s_{body}, \varphi_{body} \rangle$, where s_{body} is the tree $\{\epsilon, \text{NatNum}\}$ and

$$\varphi_{body} = \text{new}\{m\}. \text{let } \rho' = \text{app}(\text{get}_F(\rho, \text{Interv}), (\rho/I)) \\ \text{in } (m, \text{NatNum}=\rho')$$

and Σ_X represents the parameter signature of G . Here $\text{app}(\text{get}_F(\rho, \text{Interv}), \rho/I)$ stands for a functor application. The operator, $\text{get}_F(\rho, \text{Interv})$, is the functor component named *Interv* of the (unknown) realization ρ ; the operand, ρ/I , is the realization of the sub-structure named *I*.

Functor applications can generate fresh structures. For example, every application of the functor G gives rise to one fresh structure, i.e., to one structure with a fresh stamp, corresponding to the expression `struct ... end` forming the body of the functor. The realization expression `new N.φ` is used for expressing generativity. The stamps in N are bound in φ , and the semantic rules will force alpha-conversion to insure that these are replaced by “fresh” stamps when the functor is applied.

2.4 Signatures

Module interfaces are called signatures in Standard ML. A key feature is the ability to specify sharing in signatures. This is particularly important in connection with functors, as a means of stipulating sharing within the formal parameter structure. There are two forms of sharing in Standard ML: *structure sharing* and *type sharing*. The present semantics deals with structure sharing (but not with type sharing, as this requires integration with the Core language semantics.) Since functors do not have static identities, there is no notion of functor sharing specifications. As in Standard ML, a specification that two structures share is implicitly a specification that all substructures visible in both structures share as well. However, this does not imply that common functor components have the same functor signature. No attempt is made to “unify” functor signatures; indeed, there are valid signatures which cannot be matched by any real structure, because the signature imposes conflicting signatures on a specified functor. In this respect, functor specifications resemble the value specifications of Standard ML.

A functor specification can contain sharing specifications that impose sharing between the argument structure and the result structure, or between either of these and some structure declared or specified elsewhere. In that sense, sharing specifications can constrain a functor. A more detailed study of sharing, including functor sharing, is given in [14].

Formally, we represent sharing specifications by relations on structure paths, as follows. Let s be a tree. A *sharing relation* (on s) is a relation R satisfying:

1. R is an equivalence relation on $SP(s)$;
2. R is closed under structure path extension: for all sp, sp' , $strid$, if $sp R sp'$ and $sp.strid \in s$ and $sp'.strid \in s$ then $sp.strid R sp'.strid$;
3. there are no cycles in the graph obtained by collapsing R -equivalent paths into a single node.

The equivalence class containing sp is written $[sp]_R$, or just $[sp]$, when R is clear from the context. The set of equivalence classes is denoted s/R . Given any relation R on structure paths, $Cl(R)$ is the equivalence “closure” of this relation, i.e. the smallest equivalence relation on $SP(s)$ containing R .

A signature Σ is a tuple $\langle s, R, \sigma, \delta\rho, \Phi \rangle$. Here s is a tree, R is a sharing relation on s , σ is an (external) sharing map with $\text{Dom}(\sigma) \subseteq SP(s)$ mapping structure paths to stamp expressions, and Φ is a functor signature environment with $\text{Dom}(\Phi) = FP(s)$ mapping functor paths to functor signatures. The δ is a binding operator binding ρ with scope Φ , and the idea is that ρ represents the realization of a hypothetical structure matching the entire signature. It is used to express sharing between an embedded functor whose signature, Ξ , is given by Φ and a substructure specified elsewhere in the signature. This sharing is represented by a free occurrence of the stamp expression $\rho(sp)$ within Ξ . Accordingly, we require that the only free occurrences of ρ in Φ are in stamp expressions of the form $\rho(sp)$, where $sp \in s$.

The following example illustrates the roles of R and σ in representing internal and external sharing in signatures.

```

structure S = struct end;
signature SIG =
sig
  structure A: sig end
  structure B: sig end
  structure C: sig end
  sharing A = S
  sharing B = C
end;

```

The representation of this signature is $\Sigma = \langle s, R, \sigma, \delta\rho, \{\} \rangle$, where $s = \{\epsilon, A, B, C\}$, $R = Cl(\{(B, C)\})$ and $\sigma = \{A \mapsto m\}$, where m is the stamp of S .

We require that σ be consistent with R , so that it can be regarded as a partial map from s/R to stamp expressions, i.e. that if $sp R sp'$ then $\sigma(sp) = \sigma(sp')$. Furthermore, we require that the domain of σ is closed under path extension: if $sp \in \text{Dom}(\sigma)$ and $sp.strid \in s$ then $sp.strid \in \text{Dom}(\sigma)$.

2.5 Functor signatures

A functor signature Ξ takes the form $\lambda\rho : \Sigma.\Sigma_r$. Here Σ is the argument signature and Σ_r is the result signature. Write Σ_r in the form $\langle s_r, R_r, \sigma_r, \delta\rho_r, \Phi_r \rangle$. Sharing between argument and result is expressed by occurrences of stamp expressions of the form $\text{get}_S(\rho, sp)$ in σ_r and Φ_r , for some sp . We require that the only free occurrences of ρ in σ_r and Φ_r are in stamp expressions of the form $\text{get}_S(\rho, sp)$, where sp has to be a member of the tree component of Σ . This is to ensure proper propagation of sharing at functor application time.

Here is a functor specification illustrating propagation of information from the parameter of a functor to the result via the λ -bound realization variable in the functor signature.

```
functor F(X: sig structure A: sig end end):
    sig
        structure B: sig end
        sharing B = X.A
    end
```

The corresponding functor signature is $\Xi = \lambda\rho : \Sigma_X.\Sigma_R$, where $\Sigma_X = \langle\{\epsilon, A\}, I, \{\}, \delta\rho'.\{\}\rangle$ and $\Sigma_R = \langle\{\epsilon, B\}, I, \{[B] \mapsto \text{get}_S(\rho, A)\}, \delta\rho''.\{\}\rangle$ and I is the identity relation.

The next example illustrates the use of the δ -bound realization variable in expressing sharing between part of a functor and a structure specified in the same signature:

```
sig
    structure A: sig end
    functor F(X: sig structure B : sig end
              sharing B = A
              end): sig end
end
```

for which the representation is $\Sigma = \langle s, I, \sigma, \delta\rho, \{F \mapsto \Xi\} \rangle$, where $s = \{\epsilon, A, F\}$, σ is the empty map, and $\Xi = \lambda\rho' : \Sigma_X.\Sigma_R$, with $\Sigma_X = \langle\{\epsilon, B\}, I, \{[B] \mapsto \text{get}_S(\rho, A)\}, \delta\rho_1.\{\}\rangle$ and $\Sigma_R = \langle\{\epsilon\}, I, \{\}, \delta\rho_2.\{\}\rangle$.

The requirement that a δ -bound ρ only be “applied” to valid paths of the containing signature is significant for getting a well-defined notion of structure matching. Unfortunately, it also means that there is not a perfect correspondence between the present signatures and the so-called principal signatures inferred in [14]. In the latter case, one is allowed to write for example

```
sig
    structure A: sig end
    functor F(S: sig end):
        sig
            structure A': sig structure B: sig end end
            sharing A' = A
        end
end
```

in which A' is specified to share with A , although there is no specification of a B component of A outside the specification of F . Because of the requirement we are discussing, the principal signature for the above signature expression cannot be represented as a signature in the present semantics. Principal signatures that do not have such dangling components can be represented, however. Since these dangling components are easy to detect in principal signatures and could be banned without any dramatic loss in programming convenience, the two forms of signatures are not in serious conflict.

2.6 Evaluation of stamp expressions

Since we verify sharing specifications by comparing stamps, it is necessary to “evaluate” arbitrary stamp expressions to reduce them to concrete stamps. Since stamp expressions may contain realization variables, this evaluation must be performed in the context of a realization environment that binds these variables. Here are the rules defining evaluation of stamp expressions:

Stamp expressions

$\boxed{\beta \vdash \mu \Rightarrow m}$

$$\frac{}{\beta \vdash m \Rightarrow m} \quad (1)$$

$$\frac{\Sigma = \langle s, R, \sigma, \delta\rho, \Phi \rangle \quad sp \in s \quad \beta \vdash \text{get}_S(\varphi, sp) \Rightarrow m}{\beta \vdash (\text{get}_S(\varphi \downarrow \Sigma), sp) \Rightarrow m} \quad (2)$$

$$\frac{\beta \vdash \text{get}_S(\varphi, strid, sp) \Rightarrow m}{\beta \vdash \text{get}_S(\varphi / strid, sp) \Rightarrow m} \quad (3)$$

$$\frac{\beta(\rho) = \varphi \quad \beta \vdash \text{get}_S(\varphi, sp) \Rightarrow m}{\beta \vdash \text{get}_S(\rho, sp) \Rightarrow m} \quad (4)$$

$$\frac{\varphi = (\mu, -) \quad \beta \vdash \mu \Rightarrow m}{\beta \vdash \text{get}_S(\varphi, \epsilon) \Rightarrow m} \quad (5)$$

$$\frac{\varphi = (\mu, \eta) \quad \eta(strid) = \varphi' \quad \beta \vdash \text{get}_S(\varphi', sp) \Rightarrow m}{\beta \vdash \text{get}_S(\varphi, strid, sp) \Rightarrow m} \quad (6)$$

We define a function $Eval : \text{RealizationEnv} \rightarrow (\text{StampExp} \multimap \text{Stamp})$ by

$$Eval(\beta)(\mu) = \begin{cases} m & \text{if } \beta \vdash \mu \Rightarrow m \\ \text{undefined} & \text{if } \beta \vdash \mu \not\Rightarrow m \end{cases}$$

The inference system made up of the inference rules (1)–(6) is monogenic; thus the definition of $Eval$ makes sense.

Similarly, one can define rules that allow one to infer conclusions of the form $\beta \vdash \theta \Rightarrow \lambda\rho : \Sigma.\langle s, \varphi \rangle$, meaning that in the realization environment β , the value of θ is $\lambda\rho : \Sigma.\langle s, \varphi \rangle$. These rules are also monogenic and so give rise to a function $Eval : \text{RealizationEnv} \rightarrow \text{Functor} \multimap \text{Functor}$.

2.7 Elaboration of realization expressions

To evaluate stamp expressions of the form $\text{get}_S(\varphi, sp)$ that involve realization expressions, it may first be necessary to reduce the realization expression φ to a simpler form such that the rules for evaluating the stamp expression apply. The rules in this section show how to perform this reduction.

The two most interesting forms of realization expressions are $\text{app}(\theta, \varphi)$, for functor application, and $\text{new } N.\varphi$ for generativity. To handle generativity, the inference rules extend a store of currently used structure stamps each time a new stamp is picked. Thus the conclusions of the elaboration rules take the form $N, \beta \vdash \varphi \Rightarrow \varphi', N'$, and we shall always have $N' \supseteq N$.

Elaboration of functor applications involves substitution. A substitution is a finite map from realization variables to realization expressions. It can be represented by a realization environment β ; conversely, every realization environment β represents a substitution. Application of a substitution β to a term t is written $t[\beta]$.

Structure Realizations

$$N, \beta \vdash \varphi \Rightarrow \varphi', N'$$

$$\frac{N, \beta \vdash \theta \Rightarrow \lambda\rho : \Sigma.(s, \varphi) \quad N, \beta \vdash \varphi[\rho=\varphi_a \downarrow \Sigma] \Rightarrow \varphi', N'}{N, \beta \vdash \text{app}(\theta, \varphi_a) \Rightarrow \varphi', N'} \quad (7)$$

$$\frac{M \cap N = \emptyset \quad N \cup M, \beta \vdash \varphi \Rightarrow \varphi', N'}{N, \beta \vdash \text{new } M.\varphi \Rightarrow \varphi', N'} \quad (8)$$

Comment: The side-condition $M \cap N = \emptyset$ forces the stamps in M to be new. By α -conversion, M can always be chosen to satisfy the side-condition.

$$\frac{N, \beta \vdash \beta_1 \Rightarrow \beta'_1, N_1 \quad N_1, \beta \vdash \varphi[\beta'_1] \Rightarrow \varphi', N_2}{N, \beta \vdash \text{let } \beta_1 \text{ in } \varphi \Rightarrow \varphi', N_2} \quad (9)$$

$$\frac{N, \beta \vdash \varphi \Rightarrow \varphi', N'}{N, \beta \vdash (\varphi \downarrow \Sigma) \Rightarrow (\varphi' \downarrow \Sigma), N'} \quad (10)$$

Realization environments

$$N, \beta \vdash \beta \Rightarrow \beta, N$$

$$\frac{}{N, \beta \vdash \{\} \Rightarrow \{\}, N} \quad (11)$$

$$\frac{N, \beta \vdash \beta_1 \Rightarrow \beta'_1, N_1 \quad N_1, \beta \vdash \varphi_2[\beta'_1] \Rightarrow \varphi'_2, N'}{N, \beta \vdash (\beta_1, \rho=\varphi_2) \Rightarrow (\beta'_1, \rho=\varphi'_2), N'} \quad (12)$$

Rule (7) deserves some explanation. The functor $\lambda\rho : \Sigma.(s, \varphi)$ may contain free realization variables. These can be looked up in β during the elaboration of the second premise. This may seem odd in a statically scoped language, as it looks like the rule uses “dynamic binding” (β is the “call-site” environment). However, the semantics is organized in such a way that the semantic objects found for the free variables of the functor in the realization environment β at the call site are identical to the objects which were in the realization environment when the functor was declared. This is achieved, in part, by using explicit substitutions in rules (9) and (12).

3 A skeletal programming language

In this section we present a grammar and a static semantics for the skeletal language.

3.1 Grammar for programs

The grammar defining structure expressions (*strexp*) and structure-level declarations (*strdec*) is given below. (A grammar of signature expressions (*sigexp*) and specifications (*spec*) may be found in [14]).

<i>strexp</i>	<code>::= struct strdec end</code>	generative
	<code> strexp/strid</code>	structure selection
	<code> strid</code>	structure identifier
	<code> strexp:sigexp</code>	signature constraint
	<code> fp(strexp)</code>	functor application
<i>strdec</i>	<code>::= structure strid = strexp</code>	structure declaration
	<code> functor funid(strid:sigexp) = strexp</code>	functor declaration
	<code> </code>	empty
	<code> strdec; strdec</code>	sequential

3.2 Structures and environments

An environment, E , is a pair (FE, SE) , where FE is a functor environment, i.e. a finite map from FunId to terms θ representing static functors, while SE is a structure environment, i.e., a finite map from StrId to structures. Concatenation of environments, written $E_1 + E_2$ is defined in the usual way.

3.3 Structure matching

Informally speaking, a structure matches a signature if it has at least the functors and structures specified in the signature and satisfies the sharing prescribed by the signature.

Formally, let N be a stamp set, β a realization environment, and $S = \langle s, \varphi \rangle$ a structure and let $\Sigma = \langle s', R, \sigma, \delta\rho, \Phi \rangle$ be a signature. We say that S matches Σ in N and β , written $N, \beta \vdash S$ matches Σ , if

1. $s' \subseteq s$;
2. For all $sp_1, sp_2 \in s'$, if $sp_1 R sp_2$ then $\text{Eval}(\beta)(\text{get}_S(\varphi, sp_1)) = \text{Eval}(\beta)(\text{get}_S(\varphi, sp_2))$, i.e., they both exist and are equal stamps;
3. For all $sp \in s'$, if $[sp] \in \text{Dom}(\sigma)$ then $\text{Eval}(\beta)(\sigma(sp)) = \text{Eval}(\beta)(\text{get}_S(\varphi, sp))$, i.e., they both exist and are equal stamps;
4. For all $fp \in s'$, if we let $\theta = \text{Eval}(\beta)(\text{get}_F(\varphi, fp))$ and $\Xi = \Phi(fp)$, we have $N, \beta \vdash \theta$ matches $\Xi[\rho=\varphi]$;

The matching operation in item 4 is defined in Section 3.4. One of the requirements on signatures is that the only free occurrences of ρ in Ξ are of the form $\text{get}_S(\rho, sp)$, where $sp \in s'$. Therefore, only the stamps of substructures of S (not functor components of S) are relevant to the substitution in item 4.

Assuming that structure S satisfies the conditions for matching the signature Σ , the structure that results from matching S with Σ is the restriction of S to Σ , written $\text{restrict}(S, \Sigma)$ and defined as $\langle s', \varphi \downarrow \Sigma \rangle$.

3.4 Functor matching

From a signature Σ it is possible to derive a so-called *free* structure, which can be thought of as a generic representative for all the structures that match Σ . This derivation can be formalized as a relation $N, \beta \vdash \text{Free}(\Sigma) \Rightarrow S, N'$. This relation involves a stamp set because making a free structure involves picking fresh stamps for structures which are specified in Σ . One always has $N' \supseteq N$. We omit the precise definition, for lack of space, but the process is fairly straightforward, and is justified by the Principal Signatures theorem [14].

Let $\theta = \lambda \rho_1 : \Sigma'_1. \langle s''_1, \varphi''_1 \rangle$ be a functor and let $\Xi = \lambda \rho_2 : \Sigma'_2. \Sigma''_2$ be a functor signature. Write Σ'_2 in the form $\langle s'_2, R'_2, \sigma'_2, \delta\rho'_2, \Phi'_2 \rangle$. We say that θ matches Ξ in N and β , written $N, \beta \vdash \theta$ matches Ξ , if there exist $s_a, \varphi_a, N', \varphi_r$ and N'' such that

1. $N, \beta \vdash \text{Free}(\Sigma'_2) \Rightarrow \langle s_a, \varphi_a \rangle, N'$
2. $N', \beta \vdash \langle s_a, \varphi_a \rangle$ matches Σ'_1
3. $N', \beta \vdash \varphi''_1[\rho_1=\varphi_a \downarrow \Sigma'_1] \Rightarrow \varphi_r, N''$
4. $N'', \beta \vdash \langle s''_1, \varphi_r \rangle$ matches $\Sigma''_2[\rho_2=\varphi_a]$

That is, we create a free structure from Σ'_2 , apply θ to it, and check that the result matches Σ''_2 after it has been instantiated with information from the free structure.

3.5 Elaboration of structure expressions

Elaboration of structure expressions is formalized in terms of a relation

$$N, \beta^d, E \vdash \text{strexp} \Rightarrow N_1, \beta_1^d, S, \beta_1^a$$

that consumes one realization environment, β^d and produces a structure S and two realization environments, β_1^d and β_1^a . The reason is that in general we must assume that the structure expression *strexp* occurs in the body of a functor and we must achieve the effect of elaborating it formally when the functor is defined and again when the functor is applied. We introduce new realization variables to stand for all embedded functor calls, and β_1^d maps these variables to the formal realization at “definition-time” and β_1^a maps them to the unevaluated functor call expressions. The realization environment β^a can be regarded as “code” which is used in the functor body, which typically takes the form

$$\lambda \rho : \Sigma. \langle s, \text{new } N. \text{let } \beta^a \text{ in } \varphi_{\text{body}} \rangle$$

where N is the set of generative stamps of the functor and φ_{body} is the realization of the functor result. Details are found in rule 19.

Notation Rule 13 uses the following definitions, which relate to converting environments into structures. Let N be a stamp set, E be an environment and β a realization environment. Then functions *combPaths*(E) and *combReas*(E) are defined as follows. Write E in the form (FE, SE) , where $FE = \{funid_1 \mapsto \theta_1, \dots, funid_m \mapsto \theta_m\}$ and $SE = \{strid_1 \mapsto \langle s_1, \varphi_1 \rangle, \dots, strid_n \mapsto \langle s_n, \varphi_n \rangle\}$, for some m and n ($m, n \geq 0$). Then $\text{combPaths}(E) = \{\epsilon\} \cup \{funid_1, \dots, funid_m\} \cup \bigcup_{i=1}^n strid_i.s_i$ (where $strid_i.s_i$ denotes the set of paths obtained by prepending $strid_i$ to each path in s_i). Moreover, *combReas*(E) is the view

$$strid_1=\varphi_1, \dots, strid_n=\varphi_n, funid=\theta_1, \dots, funid_m=\theta_m$$

Structure Expressions

$$N, \beta^d, E \vdash \text{strexp} \Rightarrow N_1, \beta_1^d, S, \beta_1^a$$

$$\frac{\begin{array}{c} N, \beta^d, E \vdash \text{strdec} \Rightarrow N_1, \beta_1^d, E_1, \beta_1^a \quad m \notin N_1 \\ S = (\text{combPaths}(E_1), (m, \text{combReas}(E_1))) \end{array}}{N, \beta^d, E \vdash \text{struct strdec end} \Rightarrow N_1 \cup \{m\}, \beta_1^d, S, \beta_1^a} \quad (13)$$

$$\frac{N, \beta^d, E \vdash \text{strexp} \Rightarrow N_1, \beta_1^d, \langle s, \varphi \rangle, \beta_1^a \quad \text{strid} \in s}{N, \beta^d, E \vdash \text{strexp}/\text{strid} \Rightarrow N_1, \beta_1^d, \langle s/\text{strid}, \varphi/\text{strid} \rangle, \beta_1^a} \quad (14)$$

$$\frac{E(\text{strid}) = S}{N, \beta^d, E \vdash \text{strid} \Rightarrow N, \{ \}, S, \{ \}} \quad (15)$$

$$\frac{\begin{array}{c} N, \beta^d, E \vdash \text{strexp} \Rightarrow N_1, \beta_1^d, S, \beta_1^a \quad N_1, \beta^d, E \vdash \text{sigexp} \Rightarrow \Sigma \\ N_1, \beta^d + \beta_1^d \vdash S \text{ matches } \Sigma \quad S' = \text{restrict}(S, \Sigma) \end{array}}{N, \beta^d, E \vdash \text{strexp} : \text{sigexp} \Rightarrow N_1, \beta_1^d, S', \beta_1^a} \quad (16)$$

$$\frac{\begin{array}{c} E \vdash \text{fp} \Rightarrow \theta \quad \beta^d \vdash \theta \Rightarrow \lambda \rho_0 : \Sigma.(s_b, \varphi) \\ N, \beta^d, E \vdash \text{strexp} \Rightarrow N_1, \beta_1^d, S_a, \beta_1^a \quad N_1, \beta^d + \beta_1^d \vdash S_a \text{ matches } \Sigma \\ S_a = \langle s_a, \varphi_a \rangle \quad N_1, \beta^d + \beta_1^d \vdash \varphi[\rho_0 = \varphi_a \downarrow \Sigma] \Rightarrow \varphi_b, N_2 \\ \rho' \notin \text{Dom}(\beta^d + \beta_1^d) \quad \beta_2^d = \{ \rho' = \varphi_b \} \quad \beta_2^a = \{ \rho' = \text{app}(\theta, \varphi_a) \} \end{array}}{N, \beta^d, E \vdash \text{fp(strexp)} \Rightarrow N_2, \beta_2^d + \beta_2^a, \langle s_b, \rho' \rangle, \beta_1^a + \beta_2^a} \quad (17)$$

Comment: The elaboration of $\varphi[\rho_0 = \varphi_a \downarrow \Sigma]$ redoers functor applications in φ and generates fresh structures corresponding to new-bindings in φ .

Structure-level Declarations

$$N, \beta^d, E \vdash \text{strdec} \Rightarrow N_1, \beta_1^d, E_1, \beta_1^a$$

$$\frac{N, \beta^d, E \vdash \text{strexp} \Rightarrow N_1, \beta_1^d, S, \beta_1^a}{N, \beta^d, E \vdash \text{structure strid} = \text{strexp} \Rightarrow N_1, \beta_1, \{ \text{strid} \mapsto S \}, \beta_1^a} \quad (18)$$

$$\frac{\begin{array}{c} N, \beta^d, E \vdash \text{sigexp} \Rightarrow \Sigma \\ N, \beta^d \vdash \text{Free}(\Sigma) \Rightarrow S_p, N_1 \quad S_p = \langle s_p, \varphi_p \rangle \\ \rho \notin \text{Dom}(\beta^d) \\ N_1, (\beta^d, \rho = \varphi_p), E + \{ \text{strid} \mapsto \langle s_p, \rho \rangle \} \vdash \text{strexp} \Rightarrow N_2, \beta_2^d, \langle s_b, \varphi_b \rangle, \beta_2^a \\ N' = N_2 \setminus N_1 \quad \theta = \lambda \rho : \Sigma.(s_b, \text{new } N'.\text{let } \beta_2^a \text{ in } \varphi_b) \end{array}}{N, \beta^d, E \vdash \text{functor fundid(strid: sigexp)=strexp} \Rightarrow N, \{ \}, \{ \text{fundid} \mapsto \theta \}, \{ \}} \quad (19)$$

Comment: The stamp set resulting from the elaboration of the declaration is N itself, i.e., seen from outside the functor declaration, no new structures are generated. The side-condition " $\rho \notin \text{Dom}(\beta^d)$ " serves to distinguish the realization variable of strid from the realization variables of other structures, so that any free occurrence of ρ in φ_b refers to the realization of strid .

$$\frac{}{N, \beta^d, E \vdash \quad \Rightarrow N, \{ \}, \{ \}, \{ \}} \quad (20)$$

$$\frac{\begin{array}{c} N, \beta^d, E \vdash \text{strdec}_1 \Rightarrow N_1, \beta_1^d, E_1, \beta_1^a \quad N_1, \beta^d + \beta_1^d, E + E_1 \vdash \text{strdec}_2 \Rightarrow N_2, \beta_2^d, E_2, \beta_2^a \\ N, \beta^d, E \vdash \text{strdec}_1; \quad \text{strdec}_2 \Rightarrow N_2, \beta_1^d + \beta_2^d, E_1 + E_2, \beta_1^a + \beta_2^a \end{array}}{N, \beta^d, E \vdash \quad \Rightarrow N, \{ \}, \{ \}, \{ \}} \quad (21)$$

Long functor identifiers

 $E \vdash fp \Rightarrow \theta$

$$\frac{E(\text{funid}) = \theta}{E \vdash \text{funid} \Rightarrow \theta} \quad (22)$$

$$\frac{E(\text{strid}) = \langle s, \varphi \rangle}{E \vdash \text{strid}.fp \Rightarrow \text{get}_F(\varphi, fp)} \quad (23)$$

4 Conclusion

The semantics we have presented here shows that higher-order functors do not increase the complexity of the module semantics more than one would expect, and that the policy of transparent signature matching can be generalized to the higher-order case. In particular, signature matching is straightforward to check, following the definitions of the semantics.

As noted in the introduction, higher order functors behaving in accordance with this semantics have been implemented in the Standard ML of New Jersey compiler [2]. The implementation representations differ in detail from the semantic representations presented above, because of various techniques used to optimize space requirements. But taking an abstract view, there are close parallels between the semantics and the implementation.

References

1. Maria-Virginia Aponte. Extending record typing to type parametric modules with sharing. In *Twentieth Annual ACM Symp. on Principles of Prog. Languages*, pages 465–478, New York, Jan 1993. ACM Press.
2. Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Martin Wirsing, editor, *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*, New York, August 1991. Springer-Verlag. (in press).
3. Pierre Crégut. Extensions to the sml module system. Rapport de Stage d'Ingenieur Eleve des Telecommunications, November 1992.
4. Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Twenty First Annual ACM Symp. on Principles of Prog. Languages*, New York, Jan 1994. ACM Press.
5. Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Seventeenth Annual ACM Symp. on Principles of Prog. Languages*, pages 341–354, New York, Jan 1990. ACM Press.
6. Xavier Leroy. Manifest types, modules, and separate compilation. In *Twenty First Annual ACM Symp. on Principles of Prog. Languages*, New York, Jan 1994. ACM Press.
7. David MacQueen. Modules for Standard ML. In *Proc. 1984 ACM Conf. on LISP and Functional Programming*, pages 198–207, New York, 1984. ACM Press.
8. David MacQueen. Using dependent types to express modular structure. In *Thirteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 277–286, New York, Jan 1986. ACM Press.

9. Robin Milner and Mads Tofte. *Commentary on Standard ML*. MIT Press, Cambridge, Massachusetts, 1991.
10. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.
11. John C. Mitchell and Robert Harper. The essence of ML. In *Fifteenth ACM Symp. on Principles of Programming Languages*, pages 28–46, New York, 1988. ACM Press.
12. Didier Rémy. Typechecking records and variants in a natural extension of ml. In *Sixteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 77–88, New York, Jan 1989. ACM Press.
13. Donald Sannella and Andrej Tarlecki. Extended ml: Past, present, and future. Technical Report ECS-LFCS-91-138, Laboratory for Foundations of Computer Science, University of Edinburgh, 1991.
14. Mads Tofte. Principal signatures for higher-order program modules. In *Nineteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 189–199, New York, Jan 1992. ACM Press. (Extended version to appear in Journal of Functional Programming)

The PCKS-Machine: an Abstract Machine for Sound Evaluation of Parallel Functional Programs with First-Class Continuations

Luc Moreau

Institut d'Electricité Montefiore, B28. Université de Liège, Sart-Tilman, 4000 Liège,
Belgium. moreau@montefiore.ulg.ac.be

Abstract. The PCKS-machine is an abstract machine that evaluates parallel functional programs with first-class continuations. Parallelism is introduced by the construct `pcall`, which provides a fork-and-join type of parallelism. To the best of our knowledge, the PCKS-machine is the first implementation of such a language that is proved to have a transparent construct for parallelism: every program using such a construct returns the same result as in the absence of this construct. This machine is also characterised by the non-speculative invocation of continuations whose interest is illustrated in an application.

1 Introduction

The programming language Scheme [13] is often extended to parallelism by adding constructs like `future`, `fork`, or `pcall`, which explicitly indicate where evaluations can proceed in parallel [5]. These constructs are expected to be *transparent*, i.e. a program using such constructs is supposed to return the same result as in the absence of these constructs. Thus, these constructs can be regarded as annotations for parallel execution that do not change the meaning of programs. Consequently, parallel programs can be developed in two phases: first, a sequential program is written using the functional programming methodology; second, annotations for parallelism are added without changing the semantics. This approach also avoids the programmer to concentrate on parallelism-specific problems like deadlocks, race conditions, and non-determinism.

This approach has been studied at length. It began with the implementation of MultiLisp by Halstead [4] using the `future` construct; it was followed by Miller's MultiScheme [8], an extension of Scheme based on the same construct. The latter highlighted the difficulty of defining first-class continuations in a parallel setting. Katz and Weise [6] proposed a definition of the `future` construct that was suitable for first-class continuations, and they suggested a notion of *legitimacy* to guarantee the transparency of this construct; their propositions were successfully implemented by Feeley [1].

However, practice is ahead of theory in this field: two theoretical points have never been studied. First, there is no formalisation of the intuitive statement “a parallel program is expected to return the same result as in the absence of constructs for parallelism.” Second, to the best of our knowledge, no implementation was proved to be correct, probably due to a lack of formal criteria with respect to which the correctness can be established.

In a previous paper [11], we answered the first question by designing a calculus that models sequential and parallel evaluations. In this framework, a parallel

and a sequential evaluation of the same expression yield *observationally equivalent* results. Intuitively, two expressions are observationally equivalent if we can replace one by the other without being able to distinguish which one is used (as far as termination is concerned).

The goal of this paper is to answer the second question about the correctness of an implementation. Therefore, we designed the PCKS-machine, a parallel abstract machine that implements this calculus; it models a MIMD machine with a shared memory similar to those used in the mentioned implementations. We proved that this machine is sound with respect to the above notion of observational equivalence.

This paper is organised as follows. In Section 2, we give an application that uses parallelism and first-class continuations. In Section 3, we present the calculus that we previously designed and the notion of observational equivalence used to prove the correctness of our implementation. In Sections 4 and 5, we describe the PCKS-machine, an implementation of this calculus. Some properties of the machine are stated in Section 6. The correctness of the implementation is a two-step proof. First, in Section 7, we prove that there is a translation of a PCSK-machine into a term of the calculus. Second, in Section 8 we prove that any transition of the PCSK-machine preserves the observational equivalence in the calculus. We compare our approach with related work in Section 9.

2 Example

Let us consider the problem of searching and displaying the leaves of a tree that satisfy a given predicate. For efficiency reasons, we expect the leaves to be searched in parallel, but we want them to be printed in the same order as in a sequential depth-first search. Leaves are searched and displayed by the functions `search` and `display-leaves` given below.

The expression `(call/cc exp)` packages up the current continuation as an “escape procedure”, also called a reified continuation, and applies `exp` on it; this action is called *capturing* or *reifying* a continuation. Hence, in the function `search`, `exit` will be bound to the reification of the continuation that is current when entering `search`. Invoking a reified continuation on a value `v`, i.e. applying it as a regular function, resumes the computation where the continuation was captured with `v` the value of the `call/cc` expression. When a leaf satisfies the predicate `pred`, the continuation `exit` is invoked on a list containing this leaf and a reification of the current continuation; the immediate effect is the exit of the function `search` with this list as value. In the inductive case, the annotation `fork` initiates the searches in the left and right subtrees in parallel.

```
(define (search tree pred)
  (call/cc (lambda (exit)
    (letrec ((loop (lambda (tree)
      (cond ((leaf? tree) (if (pred tree)
        (call/cc (lambda (next)
          (exit (list tree next))))
        '()))
        (else (begin (fork (loop (tree->left tree)))
          (loop (tree->right tree)))))))
      (loop tree)))))
```

The function `display-leaves` begins the search with a call to the function `search`. After receiving and displaying a leaf, the function `display-leaves` invokes the received continuation to obtain the following leaf.

```
(define (display-leaves tree pred)
  (let ((a-leaf (search tree pred))) ; search for the first leaf
    (if (null? a-leaf)
        'end
        (begin (display (car a-leaf)) ; display the leaf
               ((cadr a-leaf) '()))))) ; search for the next one
```

The semantics we propose guarantees that the function `search` displays the same results in the same order as the sequential version of `search` (obtained by removing the `fork` annotation). The search of leaves would be interleaved with their displaying in the sequential version, while it is performed *speculatively* with respect to their displaying in the parallel version. The search is said to be *speculative* because the function `search` recursively traverses the tree in parallel without knowing whether the results to be found are needed.

Results are displayed as in a sequential implementation because, in the PCSK-machine, a continuation is invoked only if its invocation preserves the sequential semantics. This mechanism of invocation is said to be *non-speculative*; it is explained in Section 3 and discussed in Section 9.

3 The CPP-Calculus

The CPP-calculus [11], [3] is an extension of Plotkin’s call-by-value λ -calculus with the control operator `callcc`. The set of *terms* M of the CPP-calculus, called A_{cpp} , is defined by the following grammar.

$$M ::= V \mid (M \ M) \mid (\text{callcc } M) \mid \#_\varphi(M) \quad V ::= c \mid x \mid (\lambda x. M) \mid \langle \varphi, K[]_\varphi \rangle$$

An *application* is a juxtaposition of terms $(M \ M)$, a *callcc-application* is of the form $(\text{callcc } M)$ (the term M is called a *receiver*), and a *prompt* construct is $\#_\varphi(M)$ (with φ a *name*). A *value* V can be a constant c , a *variable* x , an *abstraction* $(\lambda x. M)$, or a *continuation point* $\langle \varphi, K[]_\varphi \rangle$, representing a reified continuation. A captured context $K[]$ and a context $C[]$ are defined by:

$$\begin{aligned} K[] &::= [] \mid K[[]M] \mid K[V[]] \mid K[\text{callcc } \lambda k. []] \mid K[\#_\varphi([])] \mid K[(\lambda v. [])V] \\ C[] &::= [] \mid C[[]M] \mid C[M[]] \mid C[\text{callcc}[]] \mid C[\#_\varphi([])] \mid C[\lambda v. []] \mid C[\langle \varphi, [] \rangle] \end{aligned}$$

In the CPP-calculus (standing for Continuation Point and Prompt), continuations have a semantics that makes them suitable for parallelism.

1. A continuation can be invoked only if one knows that it is invoked in a sequential implementation. Thus, the invocation of a continuation requires to wait for the values of all expressions that are evaluated before this invocation in a sequential implementation.
2. A continuation can be captured independently of the evaluation order.

We can observe that the invocation of a continuation seriously reduces parallelism (as opposed to the capture of a continuation). The first rule can be improved in the particular case of a *downward* continuation.

- 3 A continuation k is *downward* if k is always part of the continuation that is in effect when k is invoked. A downward continuation is always invoked in the extent of the callcc by which it was reified. In a sequential implementation, this usage of a continuation can be implemented by simply popping the stack to the desired control point. In a parallel implementation, the invocation of a downward continuation only requires to wait for the values of the expressions that are evaluated before this invocation, but are evaluated *in the extent* of the callcc by which the continuation was reified.

We can use the following strategy to detect the invocation of a downward continuation. When a continuation is reified, it is given a fresh name φ , and a mark with the same name is pushed on the stack; when a continuation with a name φ is invoked, it is said to be downward if there is a mark with the name φ in the stack. The same technique is used in the calculus; a continuation point $\langle\varphi, K[\]_\varphi\rangle$ receives a fresh name φ at the time of its creation, and a prompt $\#_\varphi(M)$ has the role of a mark with a name φ in the stack.

	$(\lambda x.M)V \rightarrow M\{V/x\}$ with V a value	(C1)
	$(ab) \rightarrow \delta(a, b)$ if this is defined	(C2)
Capture of a continuation		
	$\text{callcc } M \rightarrow \text{callcc } \lambda k. \#_\varphi(M \langle\varphi, k[\]_\varphi\rangle)$ with a fresh φ	(C3)
	$M(\text{callcc } N) \rightarrow \text{callcc } \lambda k. (\lambda f.f(N \langle p, k(f[\]_p)\rangle))M$	(C4)
	$(\text{callcc } M)N \rightarrow \text{callcc } \lambda k. (M \langle p, k([\]_p, N)\rangle)N$	(C5)
	$((\lambda f.f(\text{callcc } N))M) \rightarrow \text{callcc } \lambda k. ((\lambda f.f(N \langle p, k(f[\]_p)\rangle))M)$	(C6)
	$\#_\varphi(\text{callcc } M) \rightarrow \text{callcc } \lambda k. \#_\varphi(M \langle p, k(\#_\varphi([\]_p))\rangle)$	(C7)
	$\langle\varphi, (\text{callcc } M)\rangle \rightarrow \langle\varphi, (M \langle p, [\]_p)\rangle$	(C8)
	$\text{callcc } M \rightarrow^T M \langle p, [\]_p\rangle$	(C9)
Invocation of a continuation		
	$M(\langle\varphi, K[\]_\varphi\rangle V) \rightarrow (\langle\varphi, K[\]_\varphi\rangle V)$ with M, V values	(C10)
	$(\langle\varphi, K[\]_\varphi\rangle V)N \rightarrow (\langle\varphi, K[\]_\varphi\rangle V)$ with V a value	(C11)
	$\#_{\varphi_1}(\langle\varphi, K[\]_\varphi\rangle V) \rightarrow (\langle\varphi, K[\]_\varphi\rangle V)$ with V a value	(C12)
	$\#_\varphi(\langle\varphi, K[\]_\varphi\rangle V) \rightarrow V$ with V a value	(C13)
	$\langle\varphi, \langle\varphi_1, K_1[\]_{\varphi_1}\rangle (K_2[\]_\varphi)\rangle \rightarrow \langle\varphi, K_1[K_2[\]_\varphi]\rangle$	(C14)
	$(\langle\varphi, K[\]_\varphi\rangle V) \rightarrow^T K[V]$ with V a value	(C15)
Simplification of a prompt and of a callcc-application		
	$\#_\varphi(V) \rightarrow V$ with V a value	(C16)
	$\text{callcc } \lambda k. M \rightarrow M$ with $k \notin FV(M)$	(C17)

Fig. 1. Reduction system with continuation points and prompts: \rightarrow_{cpp}

Figure 1 displays the transitions that can be performed in the CPP-calculus. Rule C1 is Plotkin's call-by-value β -reduction, and Rule C2 is the δ -reduction. The rules C3 to C9 concern the capture of continuations. Using Rule C3, a callcc -application ($\text{callcc } M$) can be transformed into the application of the receiver

M to the continuation point $\langle \varphi, k[\]_\varphi \rangle$, which is the representation of a reified continuation. This continuation point is given a fresh name φ , which is also given to a prompt wrapping this application. Intuitively, the prompt represents a mark in the stack. The following rules are used to build, step-by-step, a representation of the continuation in the continuation point. The rules C4 to C7 have a left-hand side in which a callcc-application appears, and they have a right-hand side that is a callcc-application. Such rules are said to *bubble-up* a callcc-application from the inside of an expression towards its top level. When a callcc-application reaches the top level of an expression, Rule C9 applies the receiver on the initial continuation $\langle p, []_p \rangle$. Since this rule can only be applied at the top level, we mark it by a superscript T , and we call it a *top level rule*. According to this set of rules, a continuation can be captured when it appears in operator or in operand position of an application, or inside a prompt. Hence, the capture of a continuation is not dependent of an evaluation order.

Rules C10 to C15 describe the invocation of a continuation $\langle \varphi, K[\]_\varphi \rangle$ on a value V . The left-hand sides of these rules have an invocation of a continuation as a subexpression, and the right-hand sides of the first three equations are an invocation of the same continuation: the invocation of a continuation prunes its surrounding context. In Rule C10, the operator is required to be a value in order to preserve the left-to-right evaluation order. When a continuation with a name φ is invoked inside a prompt with the same name, this continuation is a downward continuation; by Rule C13, its invocation can be reduced to the value on which the continuation was invoked. When the invocation of the continuation reaches the top level, the top level rule C15 installs the captured context $K[\]$ and fills it with the value on which the continuation was invoked.

A *reduction* \rightarrow_{cpp} is defined as the compatible closure of the rules of Fig. 1: $M \equiv C[P] \rightarrow_{\text{cpp}} N \equiv C[Q]$, with $P \rightarrow Q$ (except C9, C15). A *computation*, $\rightarrow_{\text{cpp}}^*$, is either a reduction or an application of a top level rule:

$$M \rightarrow_{\text{cpp}}^* N = M \rightarrow_{\text{cpp}} N \cup M \xrightarrow{\text{C9}} N \cup M \xrightarrow{\text{C15}} N$$

and we note $\rightarrow_{\text{cpp}}^{**}$ its reflexive, transitive closure. The evaluation process is abstracted by a relation evalcpp: $\text{evalcpp}(M) = V$ if $M \rightarrow_{\text{cpp}}^{**} V$ with V a value.

From a programmer's point of view, two behaviours can be observed as far as termination is concerned: either a program terminates or it does not terminate. Consequently, we can say that two expressions M and N have indistinguishable behaviours, if for all contexts $C[\]$, either $C[M]$ and $C[N]$ both terminate or both do not terminate. This leads to the formal definition of observational equivalence.

Definition 1 (Observational Equivalence). $M \cong_{\text{cpp}} N$ iff for all context $C[\]$, such that $C[M]$ and $C[N]$ are programs, either both $\text{evalcpp}(C[M])$ and $\text{evalcpp}(C[N])$ are defined or both are undefined.

Parallel evaluation can be performed in the calculus using the following rule, which evaluates the subexpressions of an application in parallel.

$$M \rightarrow_{\text{cpp}} M', N \rightarrow_{\text{cpp}} N' \Rightarrow (M \ N) \rightarrow_{\text{cpp}} (M' \ N')$$

4 The PCKS-Machine: a Parallel Machine

Felleisen and Friedman [2] proposed the CEK-machine to evaluate functional programs with a control operator like callcc. We generalise the CEK-machine to

multiple processes and parallel evaluation: the PCKS-machine models a MIMD (Multiple Instruction Multiple Data) machine with a shared memory. The letters PCKS stand for Parallel machine with each process composed of a Control string and a continuation K (representing a program counter and a stack, respectively) and sharing a common Store.

A *configuration* of the PCKS-machine describes the complete state of a machine; by convention, a configuration is represented by curly letters $\mathcal{M}, \mathcal{M}_i, \dots$. A configuration \mathcal{M} is a pair $\langle P, \sigma \rangle$, composed of a set of processes P and a store σ . Each *process* is composed of a *control string* and a *continuation code*.

A *control string* is either an expression of Λ_{pcks} or the distinguished symbol \ddagger . The language accepted by the machine is noted Λ_{pcks} and is defined by the following grammar.

$$M ::= V \mid (M\ M) \mid (\text{callcc } M) \mid (\text{pcall } M\ M) \quad V ::= c \mid x \mid (\lambda x. M) \mid \langle \varphi, \kappa \rangle$$

The term $(M\ M)$ is called a *sequential-application* as opposed to the term $(\text{pcall } M\ M)$, called a *parallel-application*. For the former, the operator is evaluated before the operand, then the application is performed. For the latter, both the operator and the operand are evaluated in parallel, then the application is performed¹. Continuation points $\langle \varphi, \kappa \rangle$ are pairs composed of a name φ and a *continuation code* κ that is a p-continuation to be described below.

A *continuation code* represents the rest of the computation to be performed by a process. We distinguish between two kinds of continuation codes: **p**-continuation and **d**-continuation.

A p-continuation κ is of the form $(\text{init}) \mid (\kappa' \text{ fun } V) \mid (\kappa' \text{ arg } N) \mid (\kappa' \text{ cont}) \mid (\kappa' \text{ name } \varphi) \mid (\kappa' \text{ left } (\alpha_m, \alpha_n, N)) \mid (\kappa' \text{ right } (\alpha_m, \alpha_n))$, where κ' is also a p-continuation. The continuation code (init) represents the initial continuation. The code $(\kappa' \text{ fun } V)$ means that the expression being evaluated is the operand of an application whose operator has the value V . The code $(\kappa' \text{ arg } N)$ means that the expression being evaluated is the operator of a sequential application whose operand N is still to be evaluated. The code $(\kappa' \text{ cont})$ means that the expression being evaluated is the receiver of a callcc-application. There is also a mark that delimitates the extent of a callcc-application, but this mark is represented by the code $(\kappa' \text{ name } \varphi)$ instead of a prompt as in the CPP-calculus. The codes $(\kappa' \text{ left } (\alpha_m, \alpha_n, N))$ and $(\kappa' \text{ right } (\alpha_m, \alpha_n))$ are used when evaluating the operator and the operand of a parallel application, respectively.

A d-continuation κ is of the form $(\kappa' \text{ forked } (\alpha_i, \alpha_j)) \mid (\kappa' \text{ stop}_l \alpha) \mid (\kappa' \text{ stop}_r \alpha) \mid (\kappa' \text{ stop})$, where κ' is also a p-continuation. Such continuation codes are used in *dead* processes. The first code appears after forking processes. The second and third code appear when a process is stopped because it requires the content of an empty location α . The last code appears when a process is stopped after returning the final value.

In the above definitions, we say that κ is a *one-step extension* of κ' ; we write it $\kappa \sqsupseteq \kappa'$. The reflexive, transitive closure of \sqsupseteq , called *extension*, is written \sqsupseteq .

¹ The annotation **fork** is derived from the annotation **pcall**.

$$(\text{begin } (\text{fork } M)\ N) \equiv (\text{callcc } (\lambda k. (\text{pcall} (\text{let } ((x\ M))\ (\lambda u. u))\ (k\ N)\)))$$

By convention, lower-case letters p, p_i, \dots designate processes, and capital letters P, P_i, \dots represent sets of processes. A process p is either *active* or *dead*.

- An *active* process, $\langle M, \kappa \rangle_p$, with M an expression of Λ_{pcks} and κ a p -continuation, is a process that evaluates M with the continuation κ .
- A *dead* process, $\langle \ddot{\tau}, \kappa \rangle_p$, with κ a d -continuation, is a process that has terminated its evaluation.

The second component of a configuration is a *store* σ that binds locations to their contents. *Locations*, usually represented by $\alpha_i, \alpha_j, \dots$ letters, belong to a set of locations Loc ; they model addresses in a real computer. The content of a location can be a value of Λ_{pcks} or a data structure $[c; v]$ that we present in the following section. As usual, $\sigma(\alpha_m)$ denotes the content of the store σ at location α_m ; $\sigma(\alpha_m) \leftarrow V$ denotes the store σ after updating the location α_m with the value V . We write \perp to designate the content of an empty location.

5 Evaluation with the PCKS-Machine

Transitions of the PCKS machine are described by a relation on the set of machine configurations. We write $\mathcal{M}_i \xrightarrow{pcks} \mathcal{M}_j$ when the PCKS-configuration \mathcal{M}_i reduces to the PCKS-configuration \mathcal{M}_j . Such a *global* transition relation can be expressed in term of a *local* relation \rightarrow_p that associates one process p_k and a store σ_i to a set of processes P_{p_k} and a store σ_j : $\langle p_k, \sigma_i \rangle \rightarrow_p \langle P_{p_k}, \sigma_j \rangle$. The relation \rightarrow_p is called the *process transition* relation. Note that the relation \rightarrow_p associates a process p_k (and a store) to a set of processes P_{p_k} (and a store). There is a transition from the configuration \mathcal{M}_i to the configuration \mathcal{M}_j if there is a transition \rightarrow_p of *one* process of \mathcal{M}_i

$$\mathcal{M}_i \equiv \langle P_i, \sigma_i \rangle \xrightarrow{pcks} \mathcal{M}_j \equiv \langle P_j, \sigma_j \rangle \Leftrightarrow \exists p_k \in P_i, \langle p_k, \sigma_i \rangle \rightarrow_p \langle P_{p_k}, \sigma_j \rangle, P_j \equiv P_i \setminus \{p_k\} \cup P_{p_k}$$

We write \xrightarrow{pcks}^* for the reflexive, transitive closure of \xrightarrow{pcks} .

The process transition relation \rightarrow_p for the evaluation of sequential expressions is displayed in Fig. 2. In order to lighten the notation, a transition $\langle p_k, \sigma_i \rangle \rightarrow_p \langle P_{p_k}, \sigma_j \rangle$ is written $p_k \rightarrow_p p'_k, \sigma_i(\alpha) \leftarrow V$, when the resulting set of processes P_{p_k} contains a single process p'_k , and when the store σ_j results from an update of the store σ_i at location α .

Rules M1 to M5 concern the evaluation of sequential, purely functional expressions as in the CEK-machine (except for the substitution instead of an environment). A sequential application (MN) forces the evaluation of M before the evaluation of N by Rule M1. By Rule M5, when a value is returned to the initial continuation, the current process is stopped, and the returned value is stored in location 0, which is, by convention, the location aimed at receiving the final result of a computation.

The rules M6 to M8 concern the reification of continuations. According to Rule M6, a callcc-application begins the evaluation of its receiver with a continuation code ($\kappa \ cont$). When a value is returned to such a continuation code, this value is applied to the reification of the current continuation by Rule M7. As in the CPP-calculus (Rule C3), the continuation point is given a fresh name φ , and a continuation code ($\kappa \ name \varphi$) with the same name φ is left as a prompt in the CPP-calculus. Rule M8 corresponds to Rule C16.

$\langle(MN), \kappa\rangle_{pk} \rightarrow_p \langle M, (\kappa \text{ arg } N)\rangle_{pk}$	(M1)
$\langle V, (\kappa \text{ arg } N)\rangle_{pk} \rightarrow_p \langle N, (\kappa \text{ fun } V)\rangle_{pk}$	(M2)
$\langle V, (\kappa \text{ fun } (\lambda x.M))\rangle_{pk} \rightarrow_p \langle M\{V/x\}, \kappa\rangle_{pk}$	(M3)
$\langle b, (\kappa \text{ fun } a)\rangle_{pk} \rightarrow_p \langle \delta(a, b), \kappa\rangle_{pk}$ if $\delta(a, b)$ is defined	(M4)
$\langle V, (\text{init})\rangle_{pk} \rightarrow_p \langle \ddagger, ((\text{init}) \text{ stop})\rangle_{pk}, \sigma(0) \leftarrow V$	(M5)
$\langle \text{callcc } M, \kappa\rangle_{pk} \rightarrow_p \langle M, (\kappa \text{ cont})\rangle_{pk}$	(M6)
$\langle V, (\kappa \text{ cont})\rangle_{pk} \rightarrow_p \langle \langle\varphi, \kappa\rangle, ((\kappa \text{ name } \varphi) \text{ fun } V)\rangle_{pk}$ new φ	(M7)
$\langle V, (\kappa \text{ name } \varphi)\rangle_{pk} \rightarrow_p \langle V, \kappa\rangle_{pk}$	(M8)
$\langle V, ((\kappa \text{ fun } V') \text{ fun } \langle\varphi_0, \kappa_0\rangle)\rangle_{pk} \rightarrow_p \langle V, (\kappa \text{ fun } \langle\varphi_0, \kappa_0\rangle)\rangle_{pk}$	(M9)
$\langle V, ((\kappa \text{ cont}) \text{ fun } \langle\varphi_0, \kappa_0\rangle)\rangle_{pk} \rightarrow_p \langle V, (\kappa \text{ fun } \langle\varphi_0, \kappa_0\rangle)\rangle_{pk}$	(M10)
$\langle V, ((\kappa \text{ arg } N) \text{ fun } \langle\varphi_0, \kappa_0\rangle)\rangle_{pk} \rightarrow_p \langle V, (\kappa \text{ fun } \langle\varphi_0, \kappa_0\rangle)\rangle_{pk}$	(M11)
$\langle V, ((\kappa \text{ name } \varphi_1) \text{ fun } \langle\varphi_0, \kappa_0\rangle)\rangle_{pk} \rightarrow_p \langle V, (\kappa \text{ fun } \langle\varphi_0, \kappa_0\rangle)\rangle_{pk}$	(M12)
$\langle V, ((\text{init}) \text{ fun } \langle\varphi_0, \kappa_0\rangle)\rangle_{pk} \rightarrow_p \langle V, \kappa\rangle_{pk}$	(M13)
$\langle V, ((\kappa \text{ name } \varphi_0) \text{ fun } \langle\varphi_0, \kappa_0\rangle)\rangle_{pk} \rightarrow_p \langle V, \kappa\rangle_{pk}$	(M14)

Fig. 2. Evaluation of sequential expressions in the PCKS-machine

Rules M9 to M14 concern the invocation of a continuation point $\langle\varphi_0, \kappa_0\rangle$ on a value V . By rules M9 to M12 the current continuation code is unconditionally pruned until either an **init** or **name** continuation code is reached. On the one hand, if an **init** code is reached, Rule M13 invokes the continuation as Rule C15. On the other hand, if a **name** code with the name φ_0 is reached, we are invoking a downward continuation, and Rule M14 behaves as Rule C13.

Rule M15 introduces parallelism; the evaluation of a parallel application (**pcall** MN) creates two new processes p_i, p_j to evaluate M and N in parallel. The operator and the operand are given the continuations $(\kappa \text{ left}(\alpha_m, \alpha_n, N))$ and $(\kappa \text{ right}(\alpha_m, \alpha_n))$ respectively. A **left** continuation code indicates that the term being evaluated is an operator while a **right** continuation code indicates that the term is an operand. Both codes refer to two new empty locations α_m and α_n , which are, by construction, supposed to receive the values of M and N respectively. If α_m is empty (resp. α_n), it means that the value of the operator (resp. the operand) is not yet computed.

$$\langle \text{pcall } MN, \kappa\rangle_{pk} \rightarrow_p \{ \langle \ddagger, (\kappa \text{ forked}(\alpha_m, \alpha_n))\rangle_{pk}, \langle M, (\kappa \text{ left } (\alpha_m, \alpha_n, N))\rangle_{p_i}, \langle N, (\kappa \text{ right } (\alpha_m, \alpha_n))\rangle_{p_j} \} \text{ with fresh locations } \alpha_m, \alpha_n \quad (\text{M15})$$

Now, let us suppose that V is the value obtained by the process evaluating the operand N . In Rule M16, we consider two cases according to the content of location α_m .

- The location α_m is empty, i.e. the operator has not yet returned a value. After storing the value V in location α_n , the process evaluating the operand is stopped.
- The location α_m is not empty, i.e. both the operand and the operator have returned a value, the application can be performed with the content of α_m .

$$\begin{aligned} \langle V, (\kappa \text{ right } (\alpha_m, \alpha_n)) \rangle_{p_k} &\rightarrow_p \langle \ddagger, (\kappa \text{ stop}_r \alpha_n) \rangle_{p_k} \text{ if } \sigma(\alpha_m) = \perp, \sigma(\alpha_n) \leftarrow V & (\text{M16}) \\ \langle V, (\kappa \text{ right } (\alpha_m, \alpha_n)) \rangle_{p_k} &\rightarrow_p \langle V, (\kappa \text{ fun } V') \rangle_{p_k} \text{ if } \sigma(\alpha_m) = V', \sigma(\alpha_n) \leftarrow V \end{aligned}$$

The symmetric case concerns the evaluation of the operator M yielding a value V . Let us suppose that the location α_m is empty. Rule M17 also distinguishes between two cases according to the content of α_n . Either α_n is empty and the application cannot be performed, or α_n contains the value of the operand on which V can be applied.

$$\begin{aligned} \langle V, (\kappa \text{ left } (\alpha_m, \alpha_n, N)) \rangle_{p_k} &\rightarrow_p \langle \ddagger, (\kappa \text{ stop}_l \alpha_m) \rangle_{p_k} \text{ if } \sigma(\alpha_n) = \perp, \sigma(\alpha_m) \leftarrow V & (\text{M17}) \\ \langle V, (\kappa \text{ left } (\alpha_m, \alpha_n, N)) \rangle_{p_k} &\rightarrow_p \langle V', (\kappa \text{ fun } V) \rangle_{p_k} \text{ if } \sigma(\alpha_n) = V', \sigma(\alpha_m) \leftarrow V \end{aligned}$$

Now, let us consider the invocation of a continuation $\langle \varphi_0, \kappa_0 \rangle$ on a value V with the continuation codes **left** and **right**. According to Rule M18, a continuation code **left** can *always* be pruned regardless of the location α_n . After application of Rule M18, if the process evaluating the operand N has not obtained a value, it is said to become *speculative*, because its result is not known to be needed later. By Rule M19, the code **right** cannot be pruned if the location α_m is empty, i.e. the operator is not yet evaluated (as in Rule C10). Thus, we stop the process and store in α_n a data structure $[\langle \varphi_0, \kappa_0 \rangle; V]$, which represents the suspension of the invocation of a continuation $\langle \varphi_0, \kappa_0 \rangle$ on a value V .

$$\langle V, ((\kappa \text{ left } (\alpha_m, \alpha_n, N)) \text{ fun } \langle \varphi_0, \kappa_0 \rangle) \rangle_{p_k} \rightarrow_p \langle V, (\kappa \text{ fun } \langle \varphi_0, \kappa_0 \rangle) \rangle_{p_k} \quad (\text{M18})$$

$$\begin{aligned} \langle V, ((\kappa \text{ right } (\alpha_m, \alpha_n)) \text{ fun } \langle \varphi_0, \kappa_0 \rangle) \rangle_{p_k} &\rightarrow_p \langle \ddagger, (\kappa \text{ stop}_r \alpha_n) \rangle_{p_k} & (\text{M19}) \\ &\quad \text{if } \sigma(\alpha_m) = \perp, \sigma(\alpha_n) \leftarrow [\langle \varphi_0, \kappa_0 \rangle; V] \end{aligned}$$

$$\langle V, ((\kappa \text{ right } (\alpha_m, \alpha_n)) \text{ fun } \langle \varphi_0, \kappa_0 \rangle) \rangle_{p_k} \rightarrow_p \langle V, (\kappa \text{ fun } \langle \varphi_0, \kappa_0 \rangle) \rangle_{p_k} \text{ if } \sigma(\alpha_m) \neq \perp$$

As soon as the operator yields a value, Rule M20 helps in resuming the invocation of the continuation.

$$\begin{aligned} \langle V, (\kappa \text{ left } (\alpha_m, \alpha_n, N)) \rangle_{p_k} &\rightarrow_p \langle V', (\kappa \text{ fun } \langle \varphi_0, \kappa_0 \rangle) \rangle_{p_k} \text{ if } \sigma(\alpha_n) = [\langle \varphi_0, \kappa_0 \rangle; V'], & (\text{M20}) \\ &\quad \sigma(\alpha_m) \leftarrow V \end{aligned}$$

In the rules M17, M20, we supposed that the location α_m was empty, i.e. it was the first time a value was passed to the continuation code $(\kappa \text{ left } (\alpha_m, \alpha_n, N))$. Otherwise, if the location α_m is not empty, the continuation is said to be *multiply invoked*. The operand N must be reevaluated to preserve the sequential semantics. Hence, in Rule M21, we evaluate again the operand N .

$$\langle V, (\kappa \text{ left } (\alpha_m, \alpha_n, N)) \rangle_{p_k} \rightarrow_p \langle N, (\kappa \text{ fun } V) \rangle_{p_k} \quad \text{if } \sigma(\alpha_m) \neq \perp \quad (\text{M21})$$

By definition of the PCKS-machine, all transitions \rightarrow_p are atomic, i.e. for each rule, the operations for verifying the side-conditions, for creating processes, and for updating the store are performed in a single step.

A computation with the PCKS-machine begins with an initial configuration \mathcal{M}_{init} and terminates as soon as a final configuration \mathcal{M}_f is reached. An *initial configuration* $\mathcal{M}_{init} \equiv \{\langle M, (\text{init}) \rangle_0\}, \emptyset$ is composed of a single initial process and an empty store, where M is the program to evaluate. A *final configuration* $\mathcal{M}_f \equiv \langle P_f, \sigma_f \rangle$ is such that, the set of processes P_f contains the process $\langle \ddagger, ((\text{init})\text{stop}) \rangle_{p_k}$, and the store σ_f contains a value in the location 0.

In the following sections, we show that the machine and the calculus compute the same results. We proceed in two steps to prove such a property. First, we define a translation of a machine configuration into a term of Λ_{cpp} . Second, we prove that, for any transition of the PCKS-machine from a configuration \mathcal{M}_1 to a configuration \mathcal{M}_2 , the translation of \mathcal{M}_1 reduces to the translation of \mathcal{M}_2 in the CPP-calculus (up to observational equivalence). The translation is defined in Section 7, and the equivalence is proved in Section 8. But first, we state some properties of the machine.

6 Classes of Processes and Speculative Computation

Every time a pcall construct is reduced by Rule M15, the number of processes increases by 2, but the number of *active* processes only increases by 1. Since the number of processes with a continuation $(\kappa \text{ forked}(\alpha_m, \alpha_n))$ is equal to the number of applications of Rule M15, this number increased by one is an upper bound on the number of active processes in a configuration.

During evaluation of a parallel application (pcall $M N$), let us suppose that the evaluation of the operator M is not terminated. The process evaluating the operator performs the actions that would be performed in a sequential order, while the operand is evaluated in advance of the sequential order. A process begins a computation in advance of the sequential order if it has a continuation of the type $(\kappa \text{ right}(\alpha_m, \alpha_n))$ with an empty location α_m . It remains in advance of the sequential order until the operator gets evaluated. When the operator is evaluated, the location α_m receives a value, and the process evaluating the operand is now executing the actions that would be performed in the sequential order. (We just have to replace $(\kappa \text{ left}(\alpha_m, \alpha_n, N))$ and $(\kappa \text{ right}(\alpha_m, \alpha_n))$ by $(\kappa \text{ arg } N)$ and $(\kappa \text{ fun } V)$ respectively, where V is the content of α_m .)

Let us consider a process p_1 with a continuation κ_1 and a process $p_2 \equiv (M, \kappa_2)_{p_2}$ that is obtained by reducing p_1 (or its descendants). The process p_2 is not in advance of the sequential order with respect to κ_1 , if p_2 performs the actions that would be performed by p_1 in a sequential evaluation. In such a case, κ_2 is said to be a *sequential extension* of κ_1 , written $\kappa_2 \sqsupseteq_s \kappa_1$, satisfying

$$\kappa_2 \sqsupseteq_s \kappa_1 \iff \kappa_2 \equiv \kappa_1 \vee \begin{cases} \kappa_2 \not\equiv (\kappa' \text{ right}(\alpha_m, \alpha_n)) \wedge \kappa_2 \sqsupset \kappa' \\ \kappa_2 \equiv (\kappa' \text{ right}(\alpha_m, \alpha_n)) \wedge \sigma(\alpha_m) \neq \perp \end{cases} \text{ and } \kappa' \sqsupseteq_s \kappa_1$$

Now, we introduce the concept of *class* to specify the processes that preserve the sequential order with respect to a given continuation. We represent a class C_i by a pair $\langle \alpha_{ui}, \kappa_{ui} \rangle$, where the location α_{ui} is expected to receive (or contains) a result, and the continuation κ_{ui} waits for the result to be stored in α_{ui} . In a configuration $\mathcal{M} \equiv \langle P, \sigma \rangle$, we define the following classes: the initial class C_1 is $\langle 0, (\text{init}) \rangle$; for each process $\langle \ddagger, (\kappa \text{ forked}(\alpha_m, \alpha_n))_{p_k} \rangle$, such that $\sigma(\alpha_m) = \perp$, a new class C is defined by the location α_n and the continuation $(\kappa \text{ right}(\alpha_m, \alpha_n))$. A process $\langle M, \kappa \rangle_{p_k}$ belongs to a class $C_i \equiv \langle \alpha_{ui}, \kappa_{ui} \rangle$ if its continuation is a sequential extension of κ_{ui} , $\kappa \sqsupseteq_s \kappa_{ui}$. This notion of class specifies the number of active processes in a configuration:

Lemma 2. *Let C_1, \dots, C_n be the set of classes of the configuration $\mathcal{M} \equiv \langle P, \sigma \rangle$ with each class defined by $C_i \equiv \langle \alpha_{ui}, \kappa_{ui} \rangle$; let P_i be the set of processes belonging to class C_i . The set of processes P_i form a partition of P : $P = P_1 \cup \dots \cup P_n$, and $\forall i, j \in 1 \dots n$, $P_i \cap P_j = \emptyset$.*

Moreover, a set of processes P_i contains a single active process if and only if the content of the store at location α_{u_i} is \perp . A set of processes P_i does not contain any active process iff the content of the store at location α_{u_i} is not \perp .

It is usual to distinguish between two kinds of computations. A *mandatory* computation is a computation whose result is needed to return the final result. A *speculative* computation is a computation whose result is not known to be needed for the final result (at the time this computation is initiated), but which is launched, hoping that it will be later mandatory. We also define such notions for the PCKS-machine. Intuitively, a process p_i is said to be speculative with respect to p_j if p_j has pruned (using Rule M18) the continuation waiting for the value of process p_i .

Definition 3 (Speculative Process or Result). Let $C_i \equiv (\alpha_{u_i}, \kappa_{u_i})$ and $C_j \equiv (\alpha_{u_j}, \kappa_{u_j})$ such that $\kappa_{u_i} \equiv (\kappa \text{ right}(\alpha_{\ell_i}, \alpha_{u_i}))$ with $\kappa \sqsupseteq_s \kappa_{u_j}$.

The active process p_i of class C_i (or the result $\sigma(\alpha_i) \neq \perp$) is *speculative* with respect to a class C_j , written $p_i \notin C_j$ (or $\sigma(\alpha_i) \notin C_j$) if either

- the class C_j has no active process because location α_{u_j} is not empty, or,
- the continuation of the active process of C_j is not an extension of κ : $\kappa_j \not\sqsupseteq \kappa$.

7 Mapping a PCKS-Machine to a Term of the Calculus

We translate a configuration of the PCKS-machine into a term of the CPP-calculus. This translation is composed of three phases: process rebuilding, process merging and continuation point simplifications.

The *process-rebuilding function* has the following signature: $[\]_p : process \rightarrow process$. Intuitively, this function undoes transitions of the PCKS-machine that concerned sequential expressions:

$$\begin{aligned} [[M, \kappa \text{ cont}]]_p &= \langle (\lambda x. \text{callcc } x) M, \kappa \rangle_{p_k} & [[M, \kappa \text{ fun } F]]_p &= \langle FM, \kappa \rangle_{p_k} \\ [[M, \kappa \text{ name } \varphi]]_p &= \langle \#_\varphi(M), \kappa \rangle_{p_k} & [[M, \kappa \text{ arg } N]]_p &= \langle MN, \kappa \rangle_{p_k} \end{aligned}$$

The *process-merging function* makes a new process from processes that were forked while evaluating a pcall, and also returns a binding between a location and a value. Its signature is:

$$[., ., ., .]_{m.} : process \times process \times process \times store \rightarrow (process \times binding)$$

It takes three processes and a store; it returns a process and a binding. The binding associates a value to a location α_m allocated to receive the value of an operator by Rule M15. The process-merging function is defined by five equations:

$$\begin{aligned} &[[\ddagger, (\kappa \text{ forked}(\alpha_i, \alpha_j))]_{p_k}, \langle M_i, (\kappa \text{ left}(\alpha_i, \alpha_j, N)) \rangle_{p_i}, \langle M_j, (\kappa \text{ right}(\alpha_i, \alpha_j)) \rangle_{p_j}, \sigma]_m \quad (1) \\ &= (\langle ((\lambda f_i. (f_i M_j)) M_i), \kappa \rangle_{p_k}, \langle \alpha_i, f_i \rangle) \end{aligned}$$

$$\begin{aligned} &[[\ddagger, (\kappa \text{ forked}(\alpha_i, \alpha_j))]_{p_k}, \langle \ddagger, (\kappa \text{ stop}_l(\alpha_i)) \rangle_{p_i}, \langle M_j, (\kappa \text{ right}(\alpha_i, \alpha_j)) \rangle_{p_j}, \sigma]_m \quad (2) \\ &= (\langle (\sigma(\alpha_i) M_j), \kappa \rangle_{p_k}, \langle \alpha_i, \sigma(\alpha_i) \rangle) \end{aligned}$$

$$[[\ddagger, (\kappa \text{ forked}(\alpha_i, \alpha_j))]_{p_k}, \langle M_i, (\kappa \text{ left}(\alpha_i, \alpha_j, N)) \rangle_{p_i}, \langle \ddagger, (\kappa \text{ stop}_r \alpha_j) \rangle_{p_j}, \sigma]_m \quad (3)$$

$$= \begin{cases} (\langle ((\lambda f_i. (f_i \sigma(\alpha_j))) M_i), \kappa \rangle_{p_k}, \langle \alpha_i, f_i \rangle) & \text{if } \sigma(\alpha_i) = \perp \\ (\langle (M_i N), \kappa \rangle_{p_k}, \langle \alpha_i, \sigma(\alpha_i) \rangle) & \text{if } \sigma(\alpha_i) \neq \perp \end{cases}$$

$$\begin{aligned} &[[\ddagger, (\kappa \text{ forked}(\alpha_i, \alpha_j))]_{p_k}, \langle \ddagger, (\kappa \text{ stop}_r(\alpha_j)) \rangle_{p_i}, \langle M_j, (\kappa \text{ right}(\alpha_i, \alpha_j)) \rangle_{p_j}, \sigma]_m \quad (4) \\ &= (\langle (\sigma(\alpha_i) M_j), \kappa \rangle_{p_k}, \langle \alpha_i, \sigma(\alpha_i) \rangle) \end{aligned}$$

$$\begin{aligned} &[[\ddagger, (\kappa \text{ forked}(\alpha_i, \alpha_j))]_{p_k}, \langle M_i, (\kappa \text{ left}(\alpha_i, \alpha_j, N)) \rangle_{p_i}, \langle \ddagger, (\kappa \text{ stop}_l \alpha_i) \rangle_{p_j}, \sigma]_m \quad (5) \\ &= (\langle (M_i N), \kappa \rangle_{p_k}, \langle \alpha_i, \sigma(\alpha_i) \rangle) \end{aligned}$$

In (1), the processes created by Rule M15, with continuations $(\kappa \text{ forked } (\alpha_i, \alpha_j))$, $(\kappa \text{ left } (\alpha_i, \alpha_j, N))$, and $(\kappa \text{ right } (\alpha_i, \alpha_j))$ are translated into a process with a continuation κ and a control string $((\lambda f_i.(f_i M_j))M_i)$, which is observationally equivalent to $(M_i M_j)$. If a continuation is captured in M_j , it references the value of M_i . Thus, we introduce a parameter f_i as in Rule C4 (where f is referenced by the operator and the continuation), and we return a binding (α_i, f_i) between the location α_i and the parameter f_i , respectively intended to receive or to be bound to the value of M_i .

In (2) and (3), we proceed similarly with the continuations **left** and **right** respectively replaced by **stop_l** and **stop_r**. In (3), we take care to detect whether a value has already been passed to the **left** code in order to use the operand N in the result as in Rule M21. In (4) and (5), we consider the cases of multiple invocations.

The two phases “process rebuilding” and “process merging” are iteratively used according to the following algorithm. The translation algorithm requires a configuration of the PCKS-machine $\mathcal{M} \equiv \langle P, \sigma \rangle$, and an initially empty store σ_c . Two results are expected: *mandatory*, which is the mandatory computation, and *speculative*, which is a set of speculative computations. Initially *speculative* is an empty set and *mandatory* is undefined. Each invocation of the merging function, at step 6, extends the store σ_c with the new binding.

1. if $\langle \#, ((\text{init}) \text{ stop}) \rangle_{p_k} \in P$ then *mandatory* $\leftarrow \sigma(0)$. Proceed with $P \leftarrow P \setminus \{p_k\}$.
2. if $\langle M, (\text{init}) \rangle_{p_k} \in P$ then *mandatory* $\leftarrow M$. Proceed with $P \leftarrow P \setminus \{p_k\}$.
3. if $\exists p_i \equiv \langle M, (\kappa \text{ right } (\alpha_m, \alpha_n)) \rangle_{p_i}$, and p_i is speculative, *speculative* $\leftarrow \text{speculative} \cup \{M\}$. Proceed with $P \leftarrow P \setminus \{p_i\}$. The expression M is said to be “associated” to the class defined by $\langle \alpha_n, (\kappa \text{ right } (\alpha_n, \alpha_n)) \rangle$.
4. if $\exists \alpha_{u_i}$ that is not marked and that contains a speculative result $(\sigma(\alpha_{u_i}) \$ C_j)$, then proceed with *speculative* $\leftarrow \text{speculative} \cup \{\sigma(\alpha_{u_i})\}$, and mark α_{u_i} as visited. The content $\sigma(\alpha_{u_i})$ is said to be “associated” to the class defined by location α_{u_i} .
5. if $\exists p_i, [p_i]_p = p'_i$, then proceed with $P \leftarrow P \{p'_i/p_i\}$.
6. if $\exists p_i, p_j, p_k \in P, [p_k, p_i, p_j, \sigma]_m = (p'_k, (\alpha, v))$, then proceed with $P \leftarrow P \setminus \{p_i, p_j, p_k\} \cup \{p'_k\}$ and $\sigma_c(\alpha) \leftarrow v$.
7. stop if there is no active process in P , or no unmarked location α_{u_i} .

Resulting terms may be composed of continuation points or suspensions of invocations $[(p, \kappa_0); V]$ that remain to be translated into terms of A_{cpp} . The translation of *continuation points and suspensions of invocations* is performed by the function \mathcal{S} . The translation is straightforward for most of the terms. Suspensions of invocations are translated into the invocation of the continuation on a value, and continuation points are translated by a specialised function \mathcal{S} that maps a continuation code to a CPP context; it uses the content of the store σ_c at location α_i to translate a continuation code $(\kappa \text{ right } (\alpha_i, \alpha_j))$.

$$\begin{array}{ll}
 \overline{\lambda x.M} = \lambda x.\overline{M} & \mathcal{S}((\kappa \text{ cont}), A[\]) = \mathcal{S}(\kappa, ((\lambda x.\text{callcc } x) A[\])) \\
 \overline{x} = x & \mathcal{S}((\kappa \text{ name } \varphi), A[\]) = \mathcal{S}(\kappa, \#_\varphi(A[\])) \\
 \overline{MN} = (\overline{M} \ \overline{N}) & \mathcal{S}((\kappa \text{ fun } F), A[\]) = \mathcal{S}(\kappa, (\overline{F} \ A[\])) \\
 \overline{\text{callcc } M} = \text{callcc } (\overline{M}) & \mathcal{S}((\kappa \text{ arg } N), A[\]) = \mathcal{S}(\kappa, (A[\] \ \overline{N})) \\
 \#_\varphi(M) = \#_\varphi(\overline{M}) & \mathcal{S}((\kappa \text{ left } (\alpha_i, \alpha_j, N)), A[\]) = \mathcal{S}(\kappa, (A[\] \ \overline{N})) \\
 \overline{[(\varphi, \kappa_0); V]} = ((\varphi, \kappa_0) \ \overline{V}) & \mathcal{S}((\kappa \text{ right } (\alpha_i, \alpha_j)), A[\]) = \mathcal{S}(\kappa, (\sigma_c(\alpha_i) \ A[\])) \\
 \langle \varphi, \kappa \rangle = \langle \varphi, \mathcal{S}(\kappa, [\]) \rangle & \mathcal{S}((\text{init}), A[\]) = A[\]
 \end{array}$$

The result of the translation of a machine configuration, written $\llbracket \mathcal{M} \rrbracket$, is a set of expressions; one of them is called mandatory, while the others are called speculative. By convention, we write $\text{mandatory}(\llbracket \mathcal{M} \rrbracket)$ to denote the mandatory expression of the translation, and we write $\text{speculative}(\llbracket \mathcal{M} \rrbracket)$ to denote the set of speculative expressions of the translation.

It can be easily proved that the translation algorithm terminates. Moreover, there is one and only one translation of a machine configuration as long as this machine configuration can be reached from an initial configuration.

Lemma 4. *Let M be a program, and let $\mathcal{M}_{\text{init}} \equiv \langle \{\langle M, \text{init} \rangle_0\}, \emptyset \rangle$ be an initial configuration. Let \mathcal{M} be any configuration reachable from $\mathcal{M}_{\text{init}}$: $\mathcal{M}_{\text{init}} \xrightarrow{\text{pcks}^*} \mathcal{M}$. There exists only one mandatory expression and only one set of speculative expression for the translation of configuration \mathcal{M} , i.e. the translation is a function for configurations accessible from the initial configuration.*

If a PCKS-configuration contains a process $\langle M, \kappa \rangle_{p_k}$, the term \overline{M} appears as a subexpression of a term that results from the translation of this configuration.

Lemma 5. *Let $p_k \equiv \langle M, \kappa \rangle_{p_k}$ be a process of a PCKS-configuration \mathcal{M} . There exists an applicative context $A[\cdot]$, such that*

- if process p_k is not speculative, then $A[\overline{M}] \equiv \text{mandatory}(\llbracket \mathcal{M} \rrbracket)$,
- if process p_k is speculative, $A[\overline{M}] \in \text{speculative}(\llbracket \mathcal{M} \rrbracket)$.

where an applicative context $A[\cdot]$ is defined by:

$$A[\cdot] ::= [] \mid A[V[\cdot]] \mid A[[\cdot] M] \mid A[(\lambda f.f[\cdot])M] \mid A[\#_\varphi([\cdot])]$$

8 Equivalence of the PCKS-Machine and the CPP-Calculus

Now, we can prove that the PCKS-machine preserves the CPP-calculus.

Theorem 6. *Let M be an arbitrary program of Λ_{pcks} , and N be the corresponding program of Λ_{cpp} (obtained by removing the pcall annotations). Let \mathcal{M}_1 be the machine configuration reached from the initial configuration after n transitions, and let \mathcal{M}_2 be the machine configuration reached after $n + 1$ transitions.*

$$\mathcal{M}_{\text{init}} \equiv \langle \{\langle M, (\text{init}) \rangle_0\}, \emptyset \rangle \xrightarrow{\text{pcks}^n} \mathcal{M}_1 \xrightarrow{\text{pcks}} \mathcal{M}_2 \quad n \geq 0$$

Then, there exist two terms $N', N'' \in \Lambda_{\text{cpp}}$, such that N reduces to the mandatory term of the translation of \mathcal{M}_1 , which reduces to the mandatory term of the translation of \mathcal{M}_2 (up to observational equivalence)

$$N \xrightarrow{\text{pp}^*} N' \cong_{\text{cpp}} \text{mandatory}(\llbracket \mathcal{M}_1 \rrbracket) \xrightarrow{\text{pp}^*} N'' \cong_{\text{cpp}} \text{mandatory}(\llbracket \mathcal{M}_2 \rrbracket)$$

Moreover, if there exist two speculative terms e_1 and e_2 of the translations of \mathcal{M}_1 and \mathcal{M}_2 respectively, “associated” to the same class, then e_1 reduces to e_2 (up to observational equivalence).

Let $e_1 \in \text{speculative}(\llbracket \mathcal{M}_1 \rrbracket)$, $e_2 \in \text{speculative}(\llbracket \mathcal{M}_2 \rrbracket)$ be two terms “associated” to the same class, then there exists e'_2 such that $e_1 \xrightarrow{*_{\text{pp}}} e'_2 \cong_{\text{pp}} e_2$

It means that, for the mandatory term, any transition in the PCKS-machine corresponds to one (or more) transitions in the CPP-calculus (up to observational equivalence). For the speculative terms, a transition of the PCKS-machine preserves the observational equivalence in the CPP-calculus.

9 Related Work and Conclusion

The CEK-machine was proposed by Felleisen and Friedman [2] as a variant of Landin's SECD-machine [7]. The CEK-machine evaluates a language that is based on the control operator \mathcal{C} . When \mathcal{C} reifies a continuation, it replaces the current continuation by the initial one. Unlike `callcc`, \mathcal{C} aborts the current computation and requires to synchronise processes to capture a continuation. Although both `callcc` and \mathcal{C} are as expressive, \mathcal{C} is less suitable for parallel evaluation because it reduces parallelism.

Halstead [5, page 19] gives three criteria for the semantics of parallel constructs and continuations in a parallel Scheme. We briefly recall them here. (1) Programs using `call/cc` without constructs for parallelism should return the same results in a parallel implementation as in a sequential one. (2) Programs that use continuations exclusively in the single-use style should yield the same results as in sequential Scheme, even if a parallel construct is wrapped around arbitrary expressions. (3) Programs should yield the same results as in sequential Scheme, even if a parallel construct is wrapped around arbitrary subexpressions, with no restrictions on how continuations are used. Our implementation satisfy these three criteria for both the `pcall` and `fork` constructs.

We have based our language on the `pcall` construct. The `future` construct is different because it introduces a call-by-name parameter-passing technique. If we wish to prove the correctness of an implementation based on the `future` construct, another calculus and another notion of observational equivalence should probably be defined.

Katz and Weise [6], Feeley [1] proposed and implemented a definition of first-class continuations in a parallel Scheme with the `future` construct. Besides the construct chosen, their proposition differs from ours by the fact that continuations are invoked speculatively, i.e. without knowing whether they preserve the sequential semantics. In addition, they introduce a notion of *legitimacy* that specifies whether a result is correct. By definition, a process is said to be *legitimate* if the code it is executing would have been executed by a sequential implementation in the absence of `future`. When the evaluation begins, the initial process is given the legitimacy property. A process with the legitimacy property preserves it as long as it does not create processes. When a legitimate process p_1 forks a process p_2 (with the `future` construct), p_2 is given the legitimacy property, and p_1 loses its legitimacy. The process p_1 recovers its legitimacy when the placeholder it receives gets determined by a legitimate process.

In an implementation where continuations are invoked speculatively, one can expect more speed up, at least theoretically, although more unnecessary computations might be performed. But the example given in Section 2 is not guaranteed to return the results in the left-to-right order if continuations are invoked speculatively; the leaves are only displayed in a left-to-right order when continuations are invoked non-speculatively. In addition, Katz and Weise propose the concept of speculation barrier, which suspends *all* non-legitimate processes at a given point. This mechanism could be used to display leaves in the left-to-right order when continuations are invoked speculatively. However, the legitimacy and the speculation barrier do not appear to be able to model our continuations. Indeed, the legitimacy can be considered a global property since it requires to find a legitimacy link between the current process and the initial one. On the contrary, the non-speculative invocation of a downward continuation that we propose re-

quires to detect the legitimacy of the process invoking the continuation with respect to the process that created this continuation without knowing whether this latter process is legitimate.

To the best of our knowledge, it is the first time that an implementation of first-class continuations is proved to be correct in a parallel setting. The PCKS-machine reflects the computations that can be performed in the CPP-calculus. Consequently, this machine has the advantages of the calculus: continuations are captured independently of the evaluation order, and downward continuations are optimally invoked. But the machine has also its defaults: the machine is too cautious when invoking an upward continuation (a continuation that is not downward). However, in [10], we observed that many continuations have a limited region of effect. (Intuitively, the region of effect of a continuation is the part of the program where this continuation is accessible.) We proved that, when invoking an upward continuation, it is sufficient to wait for the values of expressions in its region of effect. Therefore, the non-speculative approach gives continuations a new role: first-class continuations can be considered a way to sequentialise operations in a parallel program; they avoid the introduction of new constructs able to sequentialise processes in programming a language.

Acknowledgements

The anonymous referees are acknowledged for their useful comments to this work.

References

1. Marc Feeley. *An Efficient and General Implementation of Futures on Large Scale Shared-Memory Multiprocessors*. PhD thesis, Brandeis University, 1993.
2. Matthias Felleisen and Daniel P. Friedman. Control Operators, the SECD-Machine and the λ -Calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217, Amsterdam, 1986. Elsevier Science Publishers.
3. Matthias Felleisen, Daniel P. Friedman, Eugene E. Kohlbecker, and Bruce Duba. A Syntactic Theory of Sequential Control. *Theor. Comp. Sci.*, 52(3):205–237, 1987.
4. Robert H. Halstead, Jr. Implementation of Multilisp : Lisp on a multiprocessor. In *Proceedings of the 1984 ACM Conference on Lisp and Functional Programming*, pages 9–17, Augustus 1984.
5. Robert H. Halstead, Jr. New ideas in parallel lisp : Language design, implementation. In T. Ito and Robert H. Halstead, editors, *Parallel Lisp : Languages and Systems. Japan.*, LNCS 441, pages 2–57. Springer-Verlag, 1990.
6. Morry Katz and Daniel Weise. Continuing Into the Future: On the Interaction of Futures and First-Class Continuations. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 176–184, June 1990.
7. P. J. Landin. The mechanical evaluation of expressions. *Comp. J.*, 6:308–320, 1964.
8. James S. Miller. *MultiScheme : A parallel processing system based on MIT Scheme*. PhD thesis, MIT, 1987.
9. Luc Moreau. An operational semantics for a parallel language with continuations. In D. Etiemble and J.-C. Syre, editors, *Parallel Architectures and Languages Europe (PARLE'92)*, LNCS 14, pages 415–430, Paris, June 1992. Springer-Verlag.
10. Luc Moreau. *Sound Evaluation of Parallel Functional Programs with First-Class Continuations*. PhD thesis, University of Liège, In preparation.
11. Luc Moreau and Daniel Ribbens. Sound Rules for Parallel Evaluation of a Functional Language with callcc. In *ACM conference on Functional Programming and Computer Architecture*, pages 125–135, Copenhagen, June 1993. ACM.
12. Gordon D. Plotkin. Call-by-name, call-by-value and the λ -calculus. *Theoretical Computer Science*, pages 125–159, 1975.
13. Jonathan Rees and William Clinger, editors. Revised⁴ Report on the Algorithmic Language Scheme. *Lisp Pointers*, 4(3):1–55, July–September 1991.

A Tiny Constraint Functional Logic Language and Its Continuation Semantics

Andy Mück, Thomas Streicher

Ludwig-Maximilians-Universität München

Leopoldstr. 11B, D-80802 München

{mueck,streiche}@informatik.uni-muenchen.de

Hendrik C.R. Lock

IBM Scientific Centre Heidelberg

Vangerowstr. 18, D-69020 Heidelberg

lock@dhdibm1.bitnet

Abstract: We present an extension of λ -calculus by logical features and constraints, which yields a minimal core language for constraint functional logic programming. We define a denotational semantics based on continuation passing style. The operational semantics of our language is given as a set of reduction rules. We prove soundness of the operational semantics w.r.t. the continuation semantics. Finally, we show how pure functional logic programs can be translated to this core language in a sound way.

1 Introduction

In recent years, many research activities focused on the combination of functional and logic programming (FLP) [3, 12, 17, 19, 22, 23, 24, 28] as well as on the combination of constraint and logic programming (CLP) [5, 6, 11, 14, 15, 16]. Similar to FLP, where the motivation is to enrich logic programming languages with functions, the motivation of our work is to combine the essential features of the constraint, functional and logic programming paradigm. For that reason, we extend λ -calculus [2] by logical features and constraints to a constraint functional logic core language (CFLP-L). We will give a continuation semantics for this language and show soundness of the operational semantics w.r.t. the continuation semantics.

The notion of constraint functional logic programming (CFLP) has been introduced by Darlington et al. [8] and further has been investigated by Lopez-Fraguas [20]. Since [8] does not even allow to use constraints in function definitions, it could not be considered as full CFLP. The work of Lopez-Fraguas starts with a lazy functional logic programming language and extends it by a constraint structure and so called “constraint conditional rewrite rules”. On denotational level, a model theoretic semantics is given. Additionally, an operational semantics is presented, which is based on narrowing. A soundness result for the operational semantics w.r.t. the declarative semantics is proved.

Our approach differs from [8] and [20]. Instead of extending a functional logic language by constraints to a full blown language, we capture the essential features of each single paradigm (constraint, functional and logic concepts) by minimal extensions of λ -calculus. These extensions are: *logical variables, non-deterministic choice, and constraints*.

Existentially quantified variables (*logical variables*) are an essential concept in logic programming. In combination with λ -calculus it is necessary to introduce an existential quantifier to declare the scope of a logical variable. Similar to logic programming a logical variable denotes the set of all its ground instances.

Non-deterministic choice between clauses is another basic concept of logic programming. We extend λ -calculus by a choice operator in order to make non-determinism explicit. Choice operators also have been studied before in connection with concurrent λ -calculi ([1, 4, 13]). However, the results of that work are less relevant for our studies, because concurrency deals with “don’t care” non-determinism, whereas in a logical setting we are interested in all the choices, i.e. we are concerned with “don’t know” non-determinism (compare [24]).

Unification, an operational concept in logic programming, is not present explicitly in our core language. We follow the idea of CLP(X) [15], which is replacing unification by constraint solving.

Similar to CLP(X), CFLP-L is parameterized by a *constraint theory*. In our approach, the constraint theory is considered as the initial model of an algebraic specification. Solving equations within this model is, on an abstract level, constraint solving. Furthermore, using algebraic specifications to describe a constraint theory has the nice side effect that, from an implementation point of view, the specification directly serves as a requirement specification for the design of the constraint solver.

As in a functional logic setting computations mostly do not terminate with a single value, but with a (possibly infinite) set of values, the denotational semantics of logical extensions of λ -calculus is usually based on powerdomains. Paterson [24] has defined a logical extension of λ -calculus with a semantics based on powerdomains. Although we have defined a powerdomain semantics for our language, too, we propose a continuation semantics of our language. First, continuations give us a useful insight to the operational behaviour. Second, continuation semantics enables later extensions by control operators [10]. The operational semantics of our language is given as set of reduction rules, where β -reduction plays the key role. Soundness of the operational semantics w.r.t. the continuation semantics is proved.

Similar to Lopez-Fraguas, we show that pure functional logic programming can be modelled within our framework. Therefore, we translate the rewrite rules of a functional logic program to CFLP-L expressions. The free constructor term algebra extracted from the functional logic program serves as the required constraint theory. Unification, then, is constraint solving in this term algebra.

Summarizing, we combine the essential features (λ -calculus, logical variables, choice operator, and constraints) of three programming paradigms (functional, logic and constraint programming) in a tiny core language. Similar to λ -calculus, which serves as *the central core* of functional languages, CFLP-L is proposed as a core of constraint functional logic languages.

The paper is organized as follows. In the next section we give the basic definitions of algebraic specifications. In section 3 we define syntax of CFLP-L. In section 4 we give a continuation semantics for CFLP-L. The operational semantics together with a soundness theorem are presented in section 5. In section 6 we show how pure functional logic programs can be translated to CFLP-L. Finally, we discuss related work.

2 Algebraic Specification of a Constraint Theory

As noted in the introduction, our language should be independent from a certain constraint theory. In order to instantiate CFLP-L with a constraint theory, we need a method to describe such theories. Since constraint theories can be considered as Σ -algebras, they can syntactically be represented by algebraic specifications.

In this section, we only give the basic definitions concerned with algebraic specifications. The reader interested in more details is referred to [27].

Definition 2.1 (Σ -Formulas)

Let X be a set of variables, $\Sigma = \{S, F\}$ be a signature with a set of sorts S and a set of function symbols F together with their arity. Let $T_\Sigma(X)$ be the set of well-formed terms over Σ and X . The set of *well-formed formulas* $WFF_\Sigma(X)$ over Σ and X is inductively defined as follows:

- i) if $t, s \in T_\Sigma(X)$ then $(t=s) \in WFF_\Sigma(X)$
- ii) if $\Phi \in WFF_\Sigma(X)$ then $\neg\Phi \in WFF_\Sigma(X)$
- iii) if $\Phi_1, \Phi_2 \in WFF_\Sigma(X)$ then $\Phi_1 \wedge \Phi_2 \in WFF_\Sigma(X)$
- iv) if $\Phi \in WFF_\Sigma(X)$ then $\forall x. \Phi \in WFF_\Sigma(X)$

A formula is called *closed* if it does not contain free variables.

Definition 2.2 (Algebraic Specification)

An *algebraic specification* $SP = (\Sigma, AX)$ consists of a signature Σ and a set of closed formulas AX over Σ . The formulas AX are called *axioms* of the specification.

Definition 2.3 (Model of an Algebraic Specification)

Let $SP = (\Sigma, AX)$. A partial Σ -algebra A is called *model* of SP , if A is term generated and if A satisfies the axioms of SP , i.e. $\forall \Phi \in AX. A \models \Phi$.

$Mod(SP)$ denotes the set of all models of a specification SP .

Let $t \in T_\Sigma(X)$ and $A \in Mod(SP)$ then $t^{A,e}$ denotes the *interpretation* of t in A under a valuation (environment) e .

Let $f: s_1 \times \dots \times s_n \rightarrow s_{n+1} \in F$. Then f^A denotes the *interpretation* of f in $A \in Mod(SP)$, i.e. f^A is a partial and strict function in $A_1 \times \dots \times A_n \rightarrow A_{n+1}$, where A_1, \dots, A_{n+1} are the *carrier sets* of the sorts s_1, \dots, s_n .

We restrict attention to term generated models, because computing in non term generated models is not possible since their elements cannot be represented. In fact, the term “CLP(IR)” is misleading because a constraint solver will calculate with representable approximations of real numbers but never will calculate with “real” real numbers.

Definition 2.4 (Set of Constructor Symbols)

Let $SP = (\Sigma, AX)$, $\Sigma = (S, F)$ be an algebraic specification. A subset $D \subseteq F$ is called a *set of constructor symbols*, if each element in each model of SP can be represented by a unique term consisting of constructors only, i.e. $\forall A \in Mod(SP) \forall a \in A \exists ! t \in T_D(\emptyset)$ such that $t^A = a$. Note that $T_D(\emptyset)$ denotes the set of all ground terms built with elements from D only (*ground constructor terms*).

Definition 2.5 (Initial Model of an Algebraic Specification)

Let $SP=(\Sigma, AX)$ be a specification. An algebra $A \in Mod(SP)$ is called *initial model* of SP , if

$\forall B \in Mod(SP)$ there exists a unique Σ -homomorphism $h: A \rightarrow B$.

We say that a specification SP is a *constraint specification* if SP has an initial model. The initial model itself is called *constraint theory*.

3 Syntax of CFLP-L

To define syntax of CFLP-L, we extend the usual formation rules of λ -calculus in order to incorporate logical variables, a choice operator and constraints.

Definition 3.1 (Syntax of CFLP-L)

Let $SP=(\Sigma, AX)$ be a constraint specification with $\Sigma=(S, F)$. Let $D \subseteq F$ be a set of constructors. The syntax of CFLP-L is given as follows.

$V \in \text{Value Expressions} \equiv x \mid \lambda x. E \mid \mu g, x. E \mid c(V, \dots, V)$	where $c \in D$
$E \in \text{Expressions} \equiv V \mid EE \mid E \amalg E \mid f(E, \dots, E) \mid \exists x. E \mid \{C\}E$	where $f \in F$
$C \in \text{Constraints} \equiv T = T \mid C, C$	
$T \in \text{Terms} \equiv V \mid f(T, \dots, T)$	where $f \in F$

Due to semantic reasons we distinguish between value-expressions and expressions. Value expressions denote expressions which are in *weak head normal form* (WHNF). Note that the restriction to term generated models of the constraint specification is necessary in order to represent each element of the constraint theory as a weak head normal form. We restrict constraints to be equations on terms. Note that predicates, if needed, may be defined as boolean valued functions within the constraint specification.

Rather than simulating recursion by the Y-combinator, we explicitly use the μ -operator to define recursive functions. Roughly speaking, $\mu g, x. M$ corresponds to an SML-like function definition $fun g x = M$.

The operator \amalg is used to express a “don’t know” non-deterministic choice between two expressions. In an expression $\exists x. M$ the abstraction operator \exists is used to declare a logical variable x with scope M . The intuitive meaning of $\exists x. M$ is the set of all $M[a/x]$ such that $a \in A$, where A is the carrier set of initial model A of SP . Accordingly, the meaning of $M \amalg N$ is the union of the meanings of M and N . Both operators set up the logic programming concepts of CFLP-L.

In order to restrict an expression M by a certain constraint C we use the notation $\{C\}M$. The meaning of $\{C\}M$ under a valuation e is equal to the meaning of M under e , if C holds under e . Otherwise the meaning of $\{C\}M$ is the empty set of results.

Note that the usual definitions of free variables and closed expression easily can be extended for our language. In the following we feel free to use these notations without explicitly defining them.

Example 3.1

Let SP be a specification of natural numbers with multiplication. The expression $\lambda x.(\exists y.\{x=2*y\}x)$ denotes a function that returns $\{x\}$ if x is even or returns the empty set (\emptyset) of results if x is odd.

Applying the function to a choice between 1,2,3,4 or 5

$$(\lambda x.(\exists y.\{x=2*y\}x))(1 \sqcup 2 \sqcup 3 \sqcup 4 \sqcup 5)$$

returns $\{2,4\}$.

Of course, applying the function to a higher order expression

$$(\lambda x.(\exists y.\{x=2*y\}x))(\lambda z.z)$$

returns \emptyset , because $\lambda z.z$ is not element of the constraint theory (here: IN).

(Note that within IN as constraint theory, the intuitive meaning of the expression $\exists x.M$ is $\{M[a/x] \mid a \in \text{IN}\}$.)

4 A Continuation Semantics for CFLP-L

Lafont, Reus and Streicher [18] have shown that for functional languages continuation semantics [10] is useful to deal with operational aspects on a denotational level. Using continuation semantics has also the advantage that later control operators can be included easily.

Let $SP=(\Sigma, AX)$ be a constraint specification, $\Sigma=(S,F)$. For simplicity, we suppose that $S=\{s\}$ consists of a single sort. Let \mathcal{A} be the carrier set of the initial model A of SP.

As example 3.1 shows, *results* of a computation in a functional logic setting are sets over \mathcal{A} . Therefore, the semantic domain \mathcal{R} of results (objects of interest) is given by

$$\mathcal{R} = \mathcal{A} \rightarrow \mathcal{O}$$

where \mathcal{O} is the Sierpinski-space (i.e. the two-element lattice consisting of \top and \perp). Since \mathcal{A} is discrete, $\mathcal{A} \rightarrow \mathcal{O}$ is isomorphic to $(\mathcal{P}(\mathcal{A}), \subseteq)$, where $\mathcal{P}(\mathcal{A})$ is the powerset of \mathcal{A} . Therefore, we feel free to use set notation in the context of \mathcal{R} . Let in the following \emptyset denote the result $\lambda a.\perp$.

Values in our computation are either elements of \mathcal{A} or mappings from values and continuations to results, where *continuations* are mappings from values to results. Thus, the semantic domain \mathcal{V} of values and the domain \mathcal{C} of continuations are given by the following equations.

$$\mathcal{V} = \mathcal{A} + (\dot{\mathcal{V}} \rightarrow \mathcal{C} \rightarrow \mathcal{R})$$

$$\mathcal{C} = \mathcal{V} \rightarrow \mathcal{R}$$

Environments are mappings from variables to values.

$$\text{Env} = \text{Var} \rightarrow \mathcal{V}$$

In the following, let

$$\text{``}f^A(v_1, \dots, v_n) \downarrow\text{''}$$

abbreviate

$$\text{``}v_1, \dots, v_n \in A \text{ and there exists } a \in A \text{ such that } a =^A f^A(v_1, \dots, v_n)\text{''}$$

The semantics of value-expressions is given by a function mapping value-expressions and environments to \mathcal{V}_\perp .

$$\boxed{\begin{aligned} \llbracket _ \rrbracket_V : V \rightarrow \text{Env} \rightarrow \mathcal{V}_\perp \\ \llbracket x \rrbracket_V e = e(x) \\ \llbracket \lambda x. M \rrbracket_V e = \lambda v. \llbracket M \rrbracket_E e[v/x] \\ \llbracket \mu g, x. M \rrbracket_V e = \text{fix}_{\mathcal{V} \rightarrow \mathcal{C} \rightarrow \mathcal{R}}(\lambda h. \lambda v. \llbracket M \rrbracket_E e[h/g, v/x]), \\ \quad \text{where } \text{fix}_{\mathcal{V} \rightarrow \mathcal{C} \rightarrow \mathcal{R}}(F) \text{ is the least fixpoint of } F. \\ \llbracket c(v_1, \dots, v_n) \rrbracket_V e = \begin{cases} c^A(\llbracket v_1 \rrbracket_V e, \dots, \llbracket v_n \rrbracket_V e), \text{ if } c^A(\llbracket v_1 \rrbracket_V e, \dots, \llbracket v_n \rrbracket_V e) \downarrow \\ \perp, \text{ otherwise} \end{cases} \end{aligned}}$$

The semantics of expressions is a function $\llbracket _ \rrbracket_E : E \rightarrow \text{Env} \rightarrow \mathcal{C} \rightarrow \mathcal{R}$.

$$\boxed{\begin{aligned} \llbracket _ \rrbracket_E : E \rightarrow \text{Env} \rightarrow \mathcal{C} \rightarrow \mathcal{R} \\ \llbracket v \rrbracket_E e k = \begin{cases} k(\llbracket v \rrbracket_V e), \text{ if } \llbracket v \rrbracket_V e \in \mathcal{V} \\ \emptyset, \text{ otherwise (note: } \llbracket v \rrbracket_V e \text{ might be } \perp) \end{cases} \\ \llbracket M N \rrbracket_E e k = \llbracket M \rrbracket_E e \lambda m. (\llbracket N \rrbracket_E e \lambda n. (m n k)) \\ \llbracket f(M_1, \dots, M_n) \rrbracket_E e k = \\ \quad \llbracket M_1 \rrbracket_E e \lambda v_1. (\llbracket M_2 \rrbracket_E e \lambda v_2. \dots (\llbracket M_n \rrbracket_E e \\ \quad \lambda v_n. \lambda a. \text{ if } f^A(v_1, \dots, v_n) \downarrow \text{ then } k(f^A(v_1, \dots, v_n) a) \dots)) \\ \llbracket \exists x. M \rrbracket_E e k = \bigcup_{a \in A} \llbracket M \rrbracket_E e[a/x] k \\ \llbracket M | N \rrbracket_E e k = \llbracket M \rrbracket_E e k \cup \llbracket N \rrbracket_E e k \\ \llbracket \{C\}M \rrbracket_E e k a = \llbracket C \rrbracket_C e \wedge \llbracket M \rrbracket_E e k a \quad (\wedge \text{ in Sierpinski-space}) \end{aligned}}$$

The semantics of values and applications is defined analogously to a continuation semantics of functional languages.

In $f(M_1, \dots, M_n)$ the evaluation order is from left to right. The continuation of the last argument first checks, whether $f^A(v_1, \dots, v_n)$ is defined in A and then continues with $f^A(v_1, \dots, v_n)$. If $f^A(v_1, \dots, v_n)$ is not defined in A then we abort computation. In such a case $\lambda a. \perp$ (\emptyset resp.) is the result. This strategy corresponds to a call-by-value evaluation strategy.

In the context of a logical variable x , the semantics of an expression M is defined as the union of all $a \in A$ of the semantics of M under the environment extension $[a/x]$. The semantics of the non-deterministic choice simply is the union of the results of the single expressions.

A constraint expression $\{C\}M$ either returns \emptyset if the constraints C are not satisfied, or it returns the results of M . Note that the above semantic equation for $\{C\}M$ is equivalent to the following one:

$$\boxed{\llbracket \{C\}M \rrbracket_E e k = \begin{cases} \llbracket M \rrbracket_E e k, \text{ if } \llbracket C \rrbracket_C e = \top \\ \emptyset, \text{ otherwise} \end{cases}}$$

The semantic equations for $\exists x.M$ and $\{C\}M$ both reflect the essence of CLP, i.e. the computation of valuations of logical variables that satisfy the constraints, and the evaluation only of those expressions which satisfy their constraints.

The semantics of constraints is given by a function mapping constraints to \mathcal{O} .

$$\boxed{\begin{array}{c} \llbracket - \rrbracket_C : C \rightarrow \text{Env} \rightarrow \mathcal{O} \\ \llbracket c_1, \dots, c_n \rrbracket_C e = \llbracket c_1 \rrbracket_{EQ} e \wedge \dots \wedge \llbracket c_n \rrbracket_{EQ} e \end{array}}$$

A sequence of constraints must be interpreted as the conjunction of all the single constraints. An equation $s=t$ holds under an environment e , if the interpretations of s and t under e are defined and equal elements in \mathcal{A} . For equality we restrict attention to elements in \mathcal{A} , because equality on non-flat predomains is not even monotonic. Therefore, an equation $t=s$ returns \perp if either t or s is a higher order term (compare example 3.1).

$$\boxed{\begin{array}{c} \llbracket - \rrbracket_{EQ} : (T \times T) \rightarrow \text{Env} \rightarrow \mathcal{O} \\ \llbracket t=s \rrbracket_{EQ} e = \begin{cases} \top, & \text{if } \llbracket t \rrbracket_T e, \llbracket s \rrbracket_T e \in \mathcal{A} \text{ and } \llbracket t \rrbracket_T e =^A \llbracket s \rrbracket_T e \\ \perp, & \text{otherwise} \end{cases} \end{array}}$$

$$\boxed{\begin{array}{c} \llbracket - \rrbracket_T : T \rightarrow \text{Env} \rightarrow \mathcal{A}_\perp \\ \llbracket v \rrbracket_T e = \begin{cases} \llbracket v \rrbracket_V e, & \text{if } \llbracket v \rrbracket_V e \in \mathcal{A} \\ \perp, & \text{otherwise} \end{cases} \\ \llbracket f(t_1, \dots, t_n) \rrbracket_T e = \begin{cases} f^A(\llbracket t_1 \rrbracket_T e, \dots, \llbracket t_n \rrbracket_T e), & \text{if } f^A(\llbracket t_1 \rrbracket_T e, \dots, \llbracket t_n \rrbracket_T e) \downarrow \\ \perp, & \text{otherwise} \end{cases} \end{array}}$$

Note that each computation starts with the empty environment e_0 and the continuation stop $\equiv \lambda v. \lambda a. (a=v)$ mapping a value v either to the singleton set $\{v\}$ if $v \in \mathcal{A}$ or to \emptyset otherwise.

Example 4.1

An example often presented in non-deterministic λ -calculi (compare [24]) is the following one:

$$(\lambda x. x+x) (2 \parallel 3)$$

The question is whether this expression is equal to $\{4,5,6\}$ or only to $\{4,6\}$. Here, the essential operational issue is whether or not the choice transported by argument x is shared in the expression $x+x$. From a denotational point of view, it is a question whether λ -abstractions denote mappings from single values to a set of results or whether λ -abstractions denote mappings from a set of values to a set of results. Since in our approach λ -abstractions denote mappings from single values to a set of results we obtain

$$\begin{aligned} \llbracket (\lambda x. x+x) (2 \parallel 3) \rrbracket_E e_0 \text{ stop} &= \llbracket (\lambda x. x+x) \rrbracket_E e_0 \lambda m. (\llbracket (2 \parallel 3) \rrbracket_E e_0 \lambda n. (m n \text{ stop})) = \\ &= \llbracket (2 \parallel 3) \rrbracket_E e_0 \lambda n. (\llbracket (\lambda x. x+x) \rrbracket_V e_0 n \text{ stop}) = \llbracket 2 \rrbracket_E e_0 \lambda n. (\llbracket x+x \rrbracket_E e_0[n/x] \text{ stop}) \cup \\ &\quad \llbracket 3 \rrbracket_E e_0 \lambda n. (\llbracket x+x \rrbracket_E e_0[n/x] \text{ stop}) = \\ &= (\llbracket x+x \rrbracket_E e_0[2/x] \text{ stop}) \cup (\llbracket x+x \rrbracket_E e_0[3/x] \text{ stop}) = \{4,6\}. \end{aligned}$$

5 Operational Semantics for CFLP-L

In the following, we give reduction rules which define the operational semantics of our language. These rules can be split into 4 groups: reduction rules, constraint rules, structural rules and distribution rules.

Let $SP=(\Sigma, AX)$ be a constraint specification, $\Sigma=(S,F)$. Let $f \in F$. Let $D \subseteq F$ be a set of constructor symbols. Let A be the initial model of SP with carrier set \mathcal{A} . Let v, v_1, \dots, v_n denote value-expressions (expressions in WHNF, see section 3), M, M_1, M_2 and N denote expressions and let C, C' denote constraints. The usual conventions of λ -calculus to avoid unintended bindings of free variables are assumed to be granted within the following rules.

Reduction Rules

- $(\lambda x.M) v \rightarrow M[x:=v]$ (note: v is a WHNF)
- $\mu g, x. M \rightarrow \lambda x. M[(\mu g, x. M)/g]$
- $f(v_1, \dots, v_n) \rightarrow t$, if $f(v_1, \dots, v_n)=t$ holds in A for some $t \in T_D(X)$

Since the continuation semantics is based on a call-by-value strategy, β -reduction applies only if the argument is evaluated to WHNF (β_{value} -reduction). The reduction rule for defined function symbols hides a call of the constraint solver, because only the constraint theory tells us how to reduce such a function application. Unfolding the μ -operator is straight forward to its treatment in a pure functional approach.

Since on an operational level neither we have the possibility to express the semantics of $\exists x. M$ by a union over all elements of A (as in the continuation semantics) nor is such a (possibly infinite) union desirable, we describe the operational behaviour of \exists in connection with a constraint expression.

Constraint Rules

Constraint Modification:

- $\exists \bar{x}. \{C\}M \rightarrow \exists \bar{x}. \{C'\}M$, if $A \models C \Leftrightarrow A \models C'$ and $FV(\{C'\}M) \subseteq FV(\{C\}M) = \{x_1, \dots, x_n\}$
- where $\exists \bar{x}$ abbreviates $\exists x_1 \dots \exists x_n$.

Constraint Propagation:

- $\exists x. \{C, x=t, C'\}M \rightarrow (\{C, C'\}M)[t/x]$, if $t \in T_\Sigma(X)$ and $x \notin \text{VAR}(t)$

Analogous to the continuation semantics, the operational semantics of our language should be independent from the constraint theory. Therefore, the constraint modification rule only describes the *essential requirement* for any constraint solver, i.e. modification of constraints in a sound and complete way. For an instantiation of our language with a certain constraint theory, this rule may be replaced by the reduction rules of a certain constraint solver (see section 6 for an example).

A constraint solver can solve constraints over its specific constraint domain only (compare example 3.1). In our approach this constraint domain is given by the constraint specification. For that reason, constraint propagation only applies if $t \in T_\Sigma(X)$ (see above). Similar constraint rules can be found in [7]. A similar propagation rule has been proposed in [19].

However, our operational semantics would even work without any propagation rule. In such a case, the reduction rules in connection with the structural and distribu-

tion rules (see below) will compute *elementary problems* of the form

$$\exists x_1. \dots \exists x_n. \{C\}t, \text{ where } t \in T_D(X).$$

which finally can be handled by the constraint solver.

The reader might be surprised that there are no failure-rules which prune computation paths with inconsistent constraints. Failure-rules would require an additional failure-constant in CFLP-L syntax. Instead of blowing up syntax and operational semantics of CFLP-L with such rules, we consider failure-rules as part of an implementation rather than as a necessary evil on semantic level.

In order to prepare expressions either for β -reduction or for constraint propagation, we need the following structural rules and distribution rules.

Structural Rules

$\{C\}(\{D\}M) \rightarrow \{C,D\}M$	$(\exists x.M)N \rightarrow \exists x.(MN)$, if $x \notin FV(N)$
$(\{C\}M)N \rightarrow \{C\}(MN)$	$\{C\}(\exists x.M) \rightarrow \exists x.(\{C\}M)$,
$v(\{C\}N) \rightarrow \{C\}(vN)$	if $x \notin VAR(C)$
$f(\dots, \{C\}M, \dots) \rightarrow \{C\}f(\dots, M, \dots)$	$v(\exists x.M) \rightarrow \exists x.(vM)$, if $x \notin FV(v)$
	$\exists x.\exists y.M \rightarrow \exists y.\exists x.M$
$f(M_1, \dots, \exists x.M_i, \dots, M_n) \rightarrow \exists x.(f(M_1, \dots, M_i, \dots, M_n))$, if $x \notin FV(M_1, \dots, M_n)$	

Distribution Rules

$\{C\}(M \parallel N) \rightarrow \{C\}M \parallel \{C\}N$	$v(N_1 \parallel N_2) \rightarrow (vN_1) \parallel (vN_2)$
$f(\dots, M_1 \parallel M_2, \dots) \rightarrow$	$\exists x.(M \parallel N) \rightarrow (\exists x.M) \parallel (\exists x.N)$
$f(\dots, M_1, \dots) \parallel f(\dots, M_2, \dots)$	$(M_1 \parallel M_2)N \rightarrow (M_1N) \parallel (M_2N)$

The distribution rules serve to distribute constraints, functions, abstractions and \exists among choices. Within a single choice the structural rules move constraints and \exists outwards. Both sets of rules interact to prepare expressions for β -reduction and constraint propagation.

Logic as well as functional logic languages usually are implemented upon several abstract machines ([12,17,19,23,26]). These abstract machines either use sophisticated graph-based [17, 19] or heap-based [12, 23, 26] term representations which have the advantage that state transitions corresponding to our structural rules can be omitted. But, the price which must be paid for saving structural rules is a complicated data structure to represent terms. Therefore, in our operational semantics we use structural rules in order to abstract from such implementation details.

Our distribution rules are closely related to choice point concepts known from implementation of logic and functional logic languages. For example, in a functional logic language, application of a function f to a choice $M_1 \parallel (M_2 \parallel M_3)$ stores $f(M_2 \parallel M_3)$ in a choice point and executes $f(M_1)$. In our operational semantics, $f(M_1 \parallel (M_2 \parallel M_3))$ reduces to $f(M_1) \parallel f(M_2 \parallel M_3)$. Roughly speaking, the first term of a choice expression can therefore be considered as actual computation, where the second term is a choice point.

Example 5.1

From an operational point of view, the question of the example 4.1 is whether or not choice expressions may be bound to a shared formal parameter.

Since in our operational semantics we have β_{value} -reduction (i.e. β -reduction applies to value expressions only) it is not possible that choice expressions are bound to formal parameters. The distribution rule

$$v(N_1 \parallel N_2) \rightarrow (vN_1) \parallel (vN_2)$$

serves to distribute an application among choices. Therefore, we obtain

$$(\lambda x. x+x) (2 \parallel 3) \xrightarrow{\text{distr.}} (\lambda x. x+x) 2 \parallel (\lambda x. x+x) 3 \xrightarrow{\beta} 4 \parallel 6.$$

The following theorem states soundness of the operational semantics w.r.t. the continuation semantics of CFLP-L. I.e. if an expression M reduces via the operational semantics to an expression N, then in the continuation semantics M and N return the same set of results for an arbitrary environment e and for an arbitrary continuation k.

Theorem 5.1 (Soundness of the Operational Semantics)

The operational semantics is sound w.r.t. the continuation semantics.

I.e. if $M \rightarrow^* N$ then $\llbracket M \rrbracket_E e k = \llbracket N \rrbracket_E e k$, for any environment e and any continuation k, where \rightarrow^* is the reflexive and transitive closure of reduction relation \rightarrow .

Proof: It is sufficient to show that all rules of the operational semantics preserve the continuation semantics. We show the proof only for β -reduction and constraint propagation. A complete proof can be found in [21].

β -Reduction:

$$\begin{aligned} \llbracket (\lambda x. M) V \rrbracket_E e k &= \llbracket \lambda x. M \rrbracket_E e \lambda m. (\llbracket V \rrbracket_E e \lambda n. (m n k)) = \\ \llbracket V \rrbracket_E e \lambda n. (\llbracket \lambda x. M \rrbracket_V e) n k &= \\ \llbracket V \rrbracket_E e \lambda n. (\llbracket M \rrbracket_E e[n/x] k) &= \llbracket M \rrbracket_E e[(\llbracket V \rrbracket_V e)/x] k = \llbracket M[V/x] \rrbracket_E e k. \end{aligned}$$

Constraint Propagation Rule:

$$\begin{aligned} \exists x. \{C, c=t, C' \} M \llbracket_E e k &= \bigcup_{a \in A} \llbracket \{C, x=t, C' \} M \rrbracket_E e[a/x] k = \\ \bigcup_{a \in A} \lambda b. (\llbracket C, x=t, C' \rrbracket_C e[a/x] \wedge \llbracket M \rrbracket_E e[a/x] k b) &= \\ \bigcup_{a \in A} \lambda b. (\llbracket C \rrbracket_C e[a/x] \wedge \llbracket x=t \rrbracket_C e[a/x] \wedge \llbracket C' \rrbracket_C e[a/x] \wedge \\ \llbracket M \rrbracket_E e[a/x] k b) &= \end{aligned}$$

(because $\llbracket x=t \rrbracket_C e[a/x]$ only holds if $x \notin \text{VAR}(t)$ and if a is equal to t^A)

$$\lambda b. (\llbracket C \rrbracket_C e[t^A/x] \wedge \llbracket C' \rrbracket_C e[t^A/x] \wedge \llbracket M \rrbracket_E e[t^A/x] k b) =$$

$$\lambda b. (\llbracket C[t/x] \rrbracket_C e \wedge \llbracket C'[t/x] \rrbracket_C e \wedge \llbracket M[t/x] \rrbracket_E e k b) =$$

$$\llbracket (\{C, C'\} M)[t/x] \rrbracket_E e k.$$

6 Pure Functional Logic Programming

In this section we will show how a certain instance of CFLP-L serves as a target language to translate pure functional logic programs.

Roughly speaking, a functional logic program consists of a signature Σ including a set D of free constructor symbols (i.e. constructor symbols not affected by the rewrite rules) and a set E of rewrite rules of the form

$$f(t_1, \dots, t_n) \rightarrow r$$

where f is a non-constructor function symbol, t_i ($1 \leq i \leq n$) are constructor terms and r is an arbitrary term over Σ .

Narrowing [9, 25] has been established as operational model for functional logic programs. In particular, narrowing tries to solve an equation eq by finding a rule $l \rightarrow r$, a non-variable subterm t of eq and a most general unifier σ such that $\sigma t = \sigma l$. Then, r is substituted for t in eq and $\sigma(eq[r/t])$ returns a new equation.

Unification is constraint solving of equations in a free term algebra. Hence, an instantiation of CFLP-L with the free constructor term algebra of a functional logic program P can serve as a target language to compile P .

As noted in section 5, in an instantiation of CFLP-L with a certain constraint theory, the constraint modification rule presented above may be replaced by the reduction rules of a certain constraint solver. For the purpose of unification, we replace this rule by the following *unification rules* which serves to reduce unification constraints.

$$\begin{aligned} \{C, c(t_1, \dots, t_n) = c(s_1, \dots, s_n), C'\}M &\rightarrow \{C, t_1 = s_1, \dots, t_n = s_n, C'\}M, \text{ } c \text{ is a constructor symbol} \\ \exists x. \{C, x = x, C'\}M &\rightarrow \exists x. (\{C, C'\}M)[t/x] \text{ (note: } x \text{ may occur in } C, C' \text{ and } M\text{)} \end{aligned}$$

Together with the constraint propagation rule (see section 5) the unification rules solve equations in the free constructor term algebra of a functional logic program P . I.e. these rules build a *specialized constraint solver* for unification.

Since detailed definition of a compiler is out of scope of this paper, let us demonstrate the compilation of a functional logic program P by means of a short example.

Example 6.1

The following functional logic program for natural numbers with addition

program	NAT
sort	Nat
cons	$0 : \text{Nat}$,
	$\text{succ} : (\text{Nat}) \text{ Nat}$
func	$\text{add} : (\text{Nat}, \text{Nat}) \text{ Nat}$
axioms	
	$\text{add}(0, y) \rightarrow y,$
	$\text{add}(\text{succ}(x), y) \rightarrow \text{succ}(\text{add}(x, y))$

end.

translates to the CFLP-L expression

$$\mu \text{ add},(x,y) . \{x=0\} y \sqcup \exists z. \{x=\text{succ}(z)\} \text{ succ}(\text{add } z \ y)$$

where the free term algebra over 0 and succ is considered as underlying constraint the-

ory specified by the constructor signature without any axioms. (Note that we extended the μ -operator for n-ary tuples of arguments.)

The rules for addition are translated to a choice between two expressions. Applying the first axiom to a term $\text{add}(m,n)$ requires unification of m with 0. Therefore, the translation the left hand side of this rule handles the constraint $\{x=0\}$ to the constraint solver. Analogously, application of the second rule requires unification of m with $\text{succ}(z)$, where z is a new logical variable. For that reason, we translate the left hand side to the constraint $\{x=\text{succ}(z)\}$, where z is existentially quantified in order to express that z is a new logical variable. The translation of the right hand sides is straight forward.

As noted above, narrowing tries to solve an equation $t_1=t_2$ over some functional logic program P. In order to solve such an equation, CFLP-L will first evaluate t and s to WHNFs (value expressions), and then pass the equation between the corresponding WHNFs to the constraint solver. For that reason, an equation $t_1=t_2$ is translated to the following CFLP-L expression which, in connection with the translation of a program, essentially implements narrowing.

$$\exists x_1. \dots \exists x_n. (\lambda t. \lambda s. (\{t=s\}t) t_1 t_2), \text{ where } \{x_1, \dots, x_n\} = \text{VAR}(t_1=t_2).$$

(Please note that expression t behind the constraint $\{t=s\}$ occurs for syntactical reasons only. It is also possible to replace this expression by a new constant SUCCESS to be defined in the constraint specification.)

Example 6.2

Finally, we show the CFLP-L computation for solving the equation $\text{add}(m,n)=2$ until the first result ($0/m, 2/n$) is computed. With

$$\text{ADD} \equiv \mu \text{ add}, x, y . \{x=0\} y \parallel \exists z. \{x=\text{succ}(z)\} \text{ succ}(\text{add} z y).$$

we obtain

$$\begin{aligned} & \exists m. \exists n. \lambda t. \lambda s. (\{t=s\}t) (\text{ADD } m \ n) 2 \rightarrow (\text{unfolding ADD}) \\ & \exists m. \exists n. \lambda t. \lambda s. (\{t=s\}t) (\{m=0\} n \parallel \\ & \exists z. \{m=\text{succ}(z)\} \text{ succ}(\text{ADD } z \ n)) 2 \rightarrow (\text{struct. rules}) \\ & \exists m. \exists n. \lambda t. \lambda s. (\{t=s\}t) (\{m=0\} n) 2 \parallel \\ & \exists m. \exists n. \lambda t. \lambda s. (\{t=s\}t) (\exists z. \{m=\text{succ}(z)\} \text{ succ}(\text{ADD } z \ n)) 2 \rightarrow (\text{struct. rules}) \\ & \exists m. \exists n. \{m=0\} (\lambda t. \lambda s. \{t=s\}t) n 2 \parallel \\ & \exists m. \exists n. \exists z. \{m=\text{succ}(z)\} \lambda t. \lambda s. (\{t=s\}t) (\text{succ}(\text{ADD } z \ n)) 2 \rightarrow \beta \\ & \exists m. \exists n. \{m=0\} (\{n=2\}n) \parallel \\ & \exists m. \exists n. \exists z. \{m=\text{succ}(z)\} \lambda t. \lambda s. (\{t=s\}t) (\text{succ}(\text{ADD } z \ n)) 2 \rightarrow (\text{struct. rule}) \\ & \exists m. \exists n. \{m=0, n=2\}n \parallel \\ & \exists m. \exists n. \exists z. \{m=\text{succ}(z)\} \lambda t. \lambda s. (\{t=s\}t) (\text{succ}(\text{ADD } z \ n)) 2 \rightarrow \dots \end{aligned}$$

7 Related Work and Conclusions

We strongly have been influenced by the work of Crossley, Mandel and Wirsing on constraint λ -calculus [7]. Constraint λ -calculus is an extension of untyped λ -calculus by constraints in order to provide a scheme for constraint functional programming sim-

ilar to CLP(X). Therefore, the constraint solver is considered as a black box, containing a decidable and representable domain of constraints. Our original motivation was to implement functional logic languages by translating them to constraint λ -calculus where a first order unification theory should fill the black box. But we soon realized that for our special needs, constraint λ -calculus lacks important logic programming concepts, namely logical variables and non-deterministic choice. Therefore, we extended constraint λ -calculus by those concepts. It turned out that this extension not only is suitable to implement functional logic programming languages, but also serves as a new approach to constraint functional logic programming. However, extending constraint λ -calculus by logical variables changes its semantics essentially.

The work of Ross Paterson [24] was the motivation for a first semantic approach to our language. Paterson extended λ -calculus with logical features and gave a denotational semantics for his language, which is based on lower powerdomains. Rather than introducing logical variables, Paterson uses a syntactic constant \mathbf{U} denoting his semantic domain $D \cong D \rightarrow PD$. In a similar semantic treatment of our language it turned out that Paterson's \mathbf{U} can in our language be expressed by $\exists x.x$. Therefore, the existential quantifier is a more "fine grain" logical feature than \mathbf{U} .

The continuation semantics of CFLP-L has been derived from a powerdomain semantics by restricting attention to observable objects, i.e. elements in the powerset of A (where A is the carrier set of the initial model of the constraint specification). Continuations have been helpful when defining the operational semantics of our language, because they allow to reason about operational questions on a denotational level [18]. On the other hand, a continuation semantics keeps our language open for later extensions by control operators.

The syntax of the functional logic core language presented in [19] is very similar, but richer because it has been designed for specifying compilation schemes for functional logic languages with different semantics. Furthermore, unification has not been treated as a separate constraint system.

As noted in the introduction, our approach differs from that of Lopez-Fraguas [20] because he extends a lazy functional logic language by constraints to a full blown language.

In this paper, we combined the *essential* features of functional, logic and constraint programming in a tiny core language. CFLP-L is intended to serve as a core of constraint functional logic programming languages. The continuation semantics presented is, as far as we know, a new approach to denotational semantics of constraint functional logic languages. We have demonstrated that narrowing can be expressed by β -reduction and constraint solving in a free term algebra.

Acknowledgements

We are very much indebted to John Crossley, Luis Mandel, and Martin Wirsing for their fruitful remarks, interesting ideas, and for many stimulating discussions. Without their work on constraint λ -calculus, this paper never would have been written.

Thanks to Rita Loogen, Michael Hanus, Heinrich Hußmann, and anonymous referees for valuable comments on a draft of this paper.

This work partially has been sponsored by the ESPRIT working group COMPASS.

References

- [1] E. Astesiano, G. Costa: Sharing in Nondeterminism. In 6th International Conference on Automata, Languages and Programming, LNCS 71, Springer Verlag 1979.
- [2] H.P. Barendregt. The Lambda Calculus. Its Syntax and Semantics. North Holland, 1984.
- [3] P.G. Bosco, E. Giovanetti, G. Levi, C. Palamedessi: A Complete Semantic Characterization of K-LEAF, a Logic Language with Partial Functions. In *Proc. 4th Symposium on Logic Programming*, San Francisco, pp. 318-327. 1987.
- [4] R.M. Burstall, J. Darlington: A Transformation System for Developing Recursive Programs. Journal of the ACM, 24(1), pp. 44-67, 1977.
- [5] J. Cohen: Constraint Logic Programming Languages. In CACM 33 (7), pp. 52-68, 1990.
- [6] A. Colmerauer: An Introduction to Prolog III. Communications of the ACM, 33 (7), pp. 52-68, 1990.
- [7] J.N. Crossley, L. Mandel, M. Wirsing: Untyped Constraint Lambda Calculus is Weakly Church Rosser, Proc. NATO-ASI Constraint Programming Summer School, Pärnu, Estonia, 1993. An extended version of this paper appears as a Research Report, Ludwig-Maximilians-Universität München, 1993.
- [8] J. Darlington, Y. Guo, H. Pull: A New Perspective in Integrating Functional and Logic Languages. Technical Report, Imperial College London, 1992.
- [9] M.J. Fay: First-Order Unification in an Equational Theory. In Proc. 4th Workshop on Automated Deduction, pp. 161-167, Austin (Texas). Academic Press, 1979.
- [10] M. Felleisen, D. Sitaram: Reasoning with Continuations II: Full Abstraction for Models of Control. In Proc ACM Conference in Lisp and Functional Programming, pp. 161-175, ACM Press, 1990.
- [11] T. Frühwirth, A. Herold, V. Küchenhoff, T. Le Provost, P. Lim, E. Monfroy, M. Wallace: Constraint Logic Programming - an Informal Introduction. Summer School in Logic Programming, Zürich, LNAI 636, pp. 3-35, Springer Verlag, 1992.
- [12] M. Hanus: Efficient Implementation of Narrowing and Rewriting. In Proc. International Workshop on Processing Declarative Knowledge, pp. 344-365. LNAI 567, Springer Verlag, 1991.
- [13] M.C.B. Hennessy, E.A. Ashcroft. A Mathematical Semantics for a Nondeterministic Typed λ -calculus. Theoretical Computer Science, 11, pp. 227-245, 1980.
- [14] P. van Hentenryck: Constraint Satisfaction in Logic Programming. MIT Press, 1989.

- [15] J. Jaffar, J.-L. Lassez: Constraint Logic Programming. In Proc. of the 14th ACM Symposium on Principles of Programming Languages, Munich (Germany), pp. 111-119. ACM Press, 1987.
- [16] J. Jaffar, S. Michaylov, P. Stuckey, R. Yap: The CLP(R) Language and System. ACM Transactions on Programming Languages and Systems, pp. 339-395, 1992.
- [17] H. Kuchen, R. Loogen, J.J. Moreno-Navarro, M. Rodriguez-Artalejo: Graph-based Implementation of a Functional Logic Language. In Proc. ESOP 90, pp. 271-290. LNCS 432, Springer Verlag, 1990.
- [18] Y. Lafont, B. Reus, Th. Streicher: From Continuation Semantics to Abstract Machines, unpublished manuscript, 1993.
- [19] H.C.R.Lock: The Implementation of Functional Logic Programming Languages, Oldenbourg-Verlag 1993.
- [20] F.J. Lopez-Fraguas: A General Scheme for Constraint Functional Logic Programming. In Proc. 3rd. International Conference on Algebraic and Logic Programming, Volterra (Italy), pp. 213-227. LNCS 632, Springer Verlag 1992.
- [21] L. Mandel, A. Mück, Th. Streicher: A New Approach to Constraint Functional Logic Programming. Forthcoming Research Report, Ludwig-Maximilians-Universität München, 1993.
- [22] D. Miller: A Logic Programming Language with Lambda Abstraction, Functional Variables and Simple Unification. Journal of Logic Programming, 1 (4), pp. 497-536, 1991.
- [23] A. Mück: Compilation of Narrowing. In Proc. of the 2nd International Workshop on Programming Language Implementation and Logic Programming, pp. 16-29. LNCS 456, Springer Verlag, 1990.
- [24] R. Paterson: A Tiny Functional Language with Logical Features, Declarative Programming, Sassbachwalden, Springer, 1991.
- [25] U.S. Reddy: Narrowing as the Operational Semantics of Functional Languages. Proc IEEE Symposium on Logic Programming, pp. 138-151, 1985.
- [26] D.H.D. Warren: An Abstract Prolog Instruction Set. Technical Note 309, SRI International, Menlo Park, California, 1993.
- [27] M. Wirsing: Algebraic Specification. In J. van Leeuwen (ed.): Handbook of Theoretical Computer Science, pp. 675-788, Elsevier Science Publishers, 1990.
- [28] D. Wolz: Design of a Compiler for Lazy Pattern Driven Narrowing. In Recent Trends in Data Type Specifications, pp. 362-379. LNCS 543, Springer Verlag, 1990.

Fully Abstract Translations and Parametric Polymorphism

Peter W. O'Hearn^{*1} and Jon G. Riecke²

¹ School of Computer & Information Science, Syracuse University

Syracuse NY 13244-4100

ohearn@top.cis.syr.edu

² AT&T Bell Laboratories

600 Mountain Avenue, Murray Hill, NJ 07974

riecke@research.att.com

Abstract. We examine three languages: call-by-name PCF; an idealized version of Algol called IA; and a call-by-name version of the functional core of ML with a parallel conditional, called PPCF+XML. Syntactic translations from PCF and IA into PPCF+XML are given and shown to be fully abstract, in the sense that they preserve and reflect observational equivalence. We believe that these results suggest the potential unifying force of Strachey's concept of parametric polymorphism.

1 Introduction

When Strachey first identified the notion of polymorphism, he immediately distinguished between two main species of polymorphic function [28]. In one form, called *ad hoc* polymorphism, a function may be applied to arguments of different types, but the algorithm may differ depending on the type. In the other form, called *parametric* polymorphism (the kind of polymorphism supported by the Girard-Reynolds polymorphic λ -calculus and the programming language Standard ML), the behaviour of a polymorphic function is determined uniformly for each instantiation of type variables. For instance, the *map* function, whose type can be written as $\forall\alpha. \forall\beta. (\alpha \rightarrow \beta) \rightarrow \text{list}[\alpha] \rightarrow \text{list}[\beta]$, works the same way across different types. Parametric polymorphism captures a form of abstraction (cf. [22, 24]): intuitively, a parametric function works in a way that does not presume knowledge of specific details of types to which it is instantiated.

Here we illustrate another connection between abstraction and parametric polymorphism: that parametric polymorphism can be used to represent in a very precise manner certain programming language features. A simple example crystallizes the general point. Consider a functional language with parametric polymorphism (e.g., the Girard-Reynolds calculus with recursion and basic arithmetic), and the polymorphic function

$$Q_P = \Lambda \text{Counter}. \lambda \text{new} : (\text{Counter} \rightarrow \text{nat}) \rightarrow \text{nat}. \\ \lambda \text{inc} : \text{Counter} \rightarrow \text{Counter}. \lambda \text{val} : \text{Counter} \rightarrow \text{nat}. P.$$

* Supported by NSF grant CCR-92110829.

We can apply Q_P to arguments that form the representation of an abstract “counter type,” e.g., $(Q_P[\text{nat}] \text{ new inc val})$ where

$$\begin{aligned}\text{new} &= \lambda p : \text{nat} \rightarrow \text{nat}. p(0) \\ \text{inc} &= \lambda n : \text{nat}. n + 1 \\ \text{val} &= \lambda n : \text{nat}. n.\end{aligned}$$

Here we are utilizing the connection between abstract types and parametric polymorphism proposed by Reynolds [22]. The application binds the type variable *Counter* to the type *nat* of natural numbers, and the formal parameters to representations of the corresponding operations of the counter type; for instance, *new* declares a new counter, initialized to 0, for use inside its argument *p*. Intuitively, the parametricity of $(\Lambda \text{Counter} \dots P)$ guarantees that the representation of the counter type can only be accessed through the given operations: one cannot, for example, apply the *inc* formal parameter to a number inside the definition of *P*, since the type of *inc* expects an argument of type *Counter*—a type that could later be bound to a functional type. But even more complex properties of the term *P* hold. For instance, a subterm of the form $(\text{new}(\lambda c : \text{Counter}. C))$ binds a new counter to *c* for use within *C*. Furthermore, only *new* can create a new counter, and the counter disappears when the execution of *C* terminates, because the type of the subterm is *nat*. Notice that this is reminiscent of the stack discipline for local variables, and that it arises *in a purely functional language!* A simple program equivalence makes the connection to local variables even more explicit. Suppose the type of Q_P is

$$\begin{aligned}\forall \text{Counter}. ((\text{Counter} \rightarrow \text{nat}) \rightarrow \text{nat}) &\rightarrow (\text{Counter} \rightarrow \text{Counter}) \\ &\rightarrow (\text{Counter} \rightarrow \text{nat}) \rightarrow \text{Counter}\end{aligned}$$

Then $(Q_P[\text{nat}] \text{ new inc val}) \equiv \Omega$ where Ω is a divergent term of type *nat*: even though a new non-divergent counter might be used in *P* (in the context of a *new* declaration), such a counter can never be returned outside the declaration, so the only counter that the application can return is Ω .

This example has to do with the “abstraction barrier” between an abstract description of a programming language and a more detailed implementation. Think of *P* as a program written in a functional language with two base types: one called *nat* with the usual operations, and one called *Counter* whose only operations are *new*, *inc*, and *val*. Two pieces of code *P*₁ and *P*₂ written in this language may only be distinguishable using the arithmetic operations and *new*, *inc*, *val*. A compiler or interpreter hides the implementation of these operations from the programmer. For instance, an interpreter could implement “deallocation” of counters by decrementing a stack pointer: even though the “old” counter may still be held in memory, a program may not access it. This is precisely what parametricity provides: a way to specify that only the operations of the base type—and no other operations derivable from the details of the implementation—may be used to operate on *Counter*.

The example illustrates an important point: reasoning about polymorphic functions in a pure functional language can provide a basis for reasoning about

certain features of local state. In this paper we develop this idea and show how to define a translation from an Algol-like language into a purely functional language with ML-style polymorphism. The translation is defined by analogy with the treatment of the “counter” example. Instead of abstracting on a single type variable *Counter*, the translation abstracts on a type variable for *each* primitive type in an Algol-like language, and passes representations of Algol base types and relevant operations (such as assignment) as arguments to a polymorphic function. In essence, the translation treats the base types and constructs of Algol as forming “higher-order” abstract data types. Our main theorem shows that that the translation is fully abstract, i.e., it preserves and reflects observational equivalence (cf. [25]).

The idea of using the “abstractness” of type variables in a polymorphic language to protect representations of types is applicable to other languages as well. For example, consider a polymorphic extension of parallel PCF, i.e., the typed λ -calculus with recursion, basic arithmetic operations, and a parallel conditional operation. Then for any term p of type

$$\forall \alpha. \alpha \rightarrow (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$$

one may obtain a function of type **bool** \rightarrow **bool** \rightarrow **bool** by instantiating α to the type **bool** of booleans and the first four arguments to the true boolean, negation, conjunction and the sequential conditional:

$$(p \text{ [bool] true not and if}) : \text{bool} \rightarrow \text{bool} \rightarrow \text{bool}.$$

For any function of type **bool** \rightarrow **bool** \rightarrow **bool** definable in sequential PCF there is a p such that this term denotes the same function. However, even though “parallel or” exists in the language, this term can never be (equivalent to) “parallel or”. Intuitively, the parametricity of p means that $(p \text{ [bool] true not and if})$ cannot use parallel facilities, because they are not definable from the given arguments. It is even possible to prove this using a model based on Reynolds’s relational approach to parametricity [24].

Based on these ideas, we define a translation from sequential PCF into a polymorphic version of parallel PCF, and again prove that the translation is fully abstract. Roughly speaking, we again treat the type of natural numbers from sequential PCF as an abstract data type. While our main interest in this translation method concerns possible applications to understanding state and related features, the PCF translation illustrates the main ideas in a simpler context: parametric polymorphism is used, as with the Algol translation, to “protect” the sequential source language from the parallelism present in the target language.

2 Sequential PCF, Idealized Algol, and PPCF+XML

In this section we define the three languages considered in this paper. In defining the languages, we often share reduction and typing rules across languages, expecting that no confusion will arise.

2.1 Sequential PCF

Our version of PCF has one base type **nat** of natural numbers. The types are

$$t ::= \text{nat} \mid t \rightarrow t.$$

We use s, t to range over types. A typing judgement is a formula of the form $\Gamma \vdash M : t$ where M is a term, t a type, and Γ is a **PCF type environment**, i.e., a finite function from variables to types. Standard rules for deriving typing judgements may be found in Table 1.

The operational semantics is given by a reduction relation $M \rightarrow N$ between terms in Table 2; this is the usual call-by-name, sequential strategy for PCF. In these rules, $M\{N/x\}$ denotes the result of substituting N for x in M with the necessary renaming of bound variables, and n denotes the n -fold application of **succ** to 0. The relation \rightarrow^* denotes the reflexive, transitive closure of \rightarrow . We define observational equivalence so that a judgement of equivalence can only be made in the presence of a type assignment. This is reasonable in a language where variables do not come tagged with types.

Definition 1. 1. $C[\cdot]$ is a PCF Γt -context if $\vdash C[M] : \text{nat}$ when $\Gamma \vdash M : t$.
 2. Suppose $\Gamma \vdash M : t$ and $\Gamma \vdash N : t$. Then $\Gamma \vdash M \equiv N$ if for all PCF Γt -contexts $C[\cdot]$, $C[M] \rightarrow^* n \Leftrightarrow C[N] \rightarrow^* n$.

Table 1 PCF Typing Rules.

$\overline{\Gamma, x : t \vdash x : t}$	$\overline{\Gamma \vdash 0 : \text{nat}}$	
$\frac{\Gamma, x : t \vdash M : s}{\Gamma \vdash (\lambda x : t.M) : t \rightarrow s}$	$\frac{\Gamma \vdash M : t \rightarrow s \quad \Gamma \vdash N : t}{\Gamma \vdash (M N) : s}$	$\frac{\Gamma \vdash M : t \rightarrow t}{\Gamma \vdash (\mathbf{Y}_t M) : t}$
$\frac{\Gamma \vdash M : \text{nat}}{\Gamma \vdash (\mathbf{succ} M) : \text{nat}}$	$\frac{\Gamma \vdash M : \text{nat}}{\Gamma \vdash (\mathbf{pred} M) : \text{nat}}$	$\frac{\Gamma \vdash M_i : \text{nat}}{\Gamma \vdash (\mathbf{ifz} M_1 M_2 M_3) : \text{nat}}$

2.2 Idealized Algol

The types of our second language, which we call IA (for Idealized Algol), are

$$\theta ::= \mathbf{exp} \mid \mathbf{loc} \mid \mathbf{comm} \mid \theta \rightarrow \theta$$

where **comm** is the type of commands, **exp** the type of expressions, and **loc** is the type of locations (or storage variables, or memory cells). For simplicity, we assume that the only storable values are natural numbers. Expressions are state-dependent values: they denote a value in a given state but do not change the state. Commands are state transformations.

Table 2 PCF Reduction Rules.

$((\lambda x. M) N) \rightarrow M\{N/x\}$	$(\text{ifz } 0 M N) \rightarrow M$
$(\mathbf{Y} M) \rightarrow (M (\mathbf{Y} M))$	$(\text{ifz } (\text{succ } n) M N) \rightarrow N$
$(\text{pred } (\text{succ } n)) \rightarrow n$	
$\frac{M \rightarrow M'}{(\text{succ } M) \rightarrow (\text{succ } M')}$	$\frac{M \rightarrow M'}{(M N) \rightarrow (M' N)}$
$\frac{M \rightarrow M'}{(\text{pred } M) \rightarrow (\text{pred } M')}$	$\frac{M \rightarrow M'}{(\text{ifz } M N P) \rightarrow (\text{ifz } M' N P)}$

The typing rules for IA appear in Table 3; π is used to denote an **IA type environment**, i.e., a finite map from variables to IA types. IA has constructs for assignment ($V := E$), dereferencing (**contents** V) and sequencing ($C_1; C_2$). Variable declarations are of the form (**new** $x = E$ in C) where C is a command, E an expression, and x a variable of location type. Variable declarations are executed by allocating a fresh location and setting the contents to the value of E in the current state, executing C where x is bound to the new location, and deallocating the location upon termination. The “newness” of local variables is the crux of the full abstraction problem for block structure and has been studied extensively (cf. [9, 13, 14]).

The reductions for IA come in two groups (see Tables 2 and 4). Purely functional reductions, the reductions found in Table 2, do not involve state, while the non-functional reductions, the reductions found in Table 4, are between configurations $[M, s]$ for M a term and s a state, i.e., a finite function from variables to numerals. For example, state is not needed for β -reduction, and so this is given by a functional reduction $(\lambda x. P)Q \rightarrow P\{Q/x\}$. In contrast, an assignment changes the state, and so we have a reduction between configurations $[x := n, s] \rightarrow [\text{skip}, s[x \mapsto n]]$, where $s[x \mapsto n]$ is the state that modifies s by mapping x to n . An IA program is a closed term of type **comm**. We observe termination of programs, and write $C \Downarrow$ to mean that $[C, \epsilon] \rightarrow^* [\text{skip}, \epsilon]$, where ϵ is the empty partial function.

Definition 2. 1. $C[\cdot]$ is an IA $\pi\theta$ -context if $\vdash C[M] : \text{comm}$ when $\pi \vdash M : \theta$.
 2. Suppose that $\pi \vdash M : \theta$ and $\pi \vdash N : \theta$. Then $\pi \vdash M \equiv N$ if for all IA $\pi\theta$ -contexts $C[\cdot]$, $C[M] \Downarrow \Leftrightarrow C[N] \Downarrow$.

2.3 PPCF+XML

The target language for our translations, called PPCF+XML, is PCF extended with a parallel conditional and with an explicitly-typed version of Milner’s polymorphic let [11]. The type system is essentially the XML type system of Mitchell and Harper [12, 5].

Table 3 IA Typing Rules.

$\frac{}{\pi, x : \theta \vdash x : \theta}$	$\frac{}{\pi \vdash 0 : \mathbf{exp}}$	$\frac{}{\pi \vdash \mathbf{skip} : \mathbf{comm}}$
$\frac{\pi, x : \theta \vdash M : \theta'}{\pi \vdash (\lambda x : \theta. M) : \theta \rightarrow \theta'}$	$\frac{\pi \vdash M : \theta \rightarrow \theta' \quad \pi \vdash N : \theta}{\pi \vdash (M N) : \theta'}$	$\frac{\pi \vdash M : \theta \rightarrow \theta}{\pi \vdash (\mathbf{Y}_\theta M) : \theta}$
$\frac{\pi \vdash M : \mathbf{exp}}{\pi \vdash (\mathbf{succ} M) : \mathbf{exp}}$	$\frac{\pi \vdash M : \mathbf{exp}}{\pi \vdash (\mathbf{pred} M) : \mathbf{exp}}$	
$\frac{\pi \vdash C_1 : \mathbf{comm} \quad \pi \vdash C_2 : \mathbf{comm}}{\pi \vdash C_1; C_2 : \mathbf{comm}}$	$\frac{\pi \vdash M : \mathbf{loc} \quad \pi \vdash E : \mathbf{exp}}{\pi \vdash M := E : \mathbf{comm}}$	
$\frac{\pi, z : \mathbf{loc} \vdash C : \mathbf{comm} \quad \pi \vdash E : \mathbf{exp}}{\pi \vdash \mathbf{new} z = E \text{ in } C : \mathbf{comm}}$	$\frac{\pi \vdash M : \mathbf{loc}}{\pi \vdash \mathbf{contents} M : \mathbf{exp}}$	
	$\frac{\pi \vdash M : \mathbf{exp} \quad \pi \vdash N : B \quad \pi \vdash P : B}{\pi \vdash (\mathbf{ifz}_B M N P) : B}$ <small>B a base type</small>	

Table 4 Additional Reduction Rules for IA.

$\frac{M \rightarrow N}{[M, s] \rightarrow [N, s]}$	
$[(\mathbf{contents} x), s] \rightarrow [s(x), s]$	$[(\mathbf{skip}; C), s] \rightarrow [C, s]$
$[(x := n), s] \rightarrow [\mathbf{skip}, s[x \mapsto n]]$	$[(\mathbf{new} x = m \text{ in } \mathbf{skip}), s] \rightarrow [\mathbf{skip}, s]$
$\frac{[M, s] \rightarrow [M', s]}{[(\mathbf{contents} M), s] \rightarrow [(\mathbf{contents} M'), s]}$	$\frac{[C_1, s] \rightarrow [C'_1, s']}{{[(C_1; C_2), s] \rightarrow [(C'_1; C_2), s'] }}$
$\frac{[E, s] \rightarrow [E', s]}{[(\mathbf{new} x = E \text{ in } C), s] \rightarrow [(\mathbf{new} x = E' \text{ in } C), s]}$	$\frac{[M, s] \rightarrow [M', s]}{[(M := E), s] \rightarrow [(M' := E), s]}$
$\frac{[C, s[x \mapsto m]] \rightarrow [C', s'[x \mapsto n]]}{[(\mathbf{new} x = m \text{ in } C), s] \rightarrow [(\mathbf{new} x = n \text{ in } C'), s']}$	$\frac{[E, s] \rightarrow [E', s]}{[(x := E), s] \rightarrow [(x := E'), s]}$

PPCF+XML has two kinds of type, called *types* and *type schemes*:

$$\begin{array}{ll} t ::= \alpha \mid \mathbf{nat} \mid t \rightarrow t & \text{(types)} \\ T ::= t \mid \forall \alpha. T & \text{(type schemes)} \end{array}$$

We use s, t to range over types, S, T to range over type schemes, and α, β to range over type variables (which are distinct from ordinary program variables). Note that types are allowed to contain type variables, though no instances of \forall .

The grammar of terms is essentially that for PCF extended with four constructs: $(\lambda \alpha. M)$ for abstraction on a type variable, $(M t)$ for application of a

polymorphic term to a type (not a type scheme), an explicitly-typed version of Milner's let construct for binding variables of polymorphic type, and a parallel conditional of the form $(\text{pifz } M N P)$ where N and P must be of type nat. A typing judgement in PPCF+XML is of the form $\Delta \vdash M : T$, where now Δ is a finite function from variables to type schemes, and T is a type scheme. The typing rules are given in Tables 1 and 5. (Notes: $\text{FTV}(\Delta)$ is the set of free type variables in type schemes assigned by Δ , and $M\{t/\alpha\}$ denotes the substitution of a type for a type variable in M .) The operational semantics (Tables 2 and 6) extends that of PCF with rules for pifz, β -reduction for types, and a β -reduction rule for reducing let's.

To define equivalence in the polymorphic language we must keep track of free type variables in typing judgements. We write $X; \Delta \vdash M : T$ when $\Delta \vdash M : T$ is derivable and X is a set of type variables containing those free in T , M , and Δ . Thus, $\vdash M : T$ means that there are no type variables free in M or T .

- Definition 3.** 1. $C[\cdot]$ is a PPCF+XML $X \Delta T$ -context if $\vdash C[M] : \text{nat}$ whenever $X; \Delta \vdash M : T$.
 2. Suppose that $X; \Delta \vdash M : T$ and $X; \Delta \vdash N : T$. Then $X; \Delta \vdash M \equiv N$ if for all PPCF+XML $X \Delta T$ -contexts $C[\cdot]$, $C[M] \rightarrow^* n \Leftrightarrow C[N] \rightarrow^* n$.

Table 5 Additional Typing Rules for PPCF+XML.

$\frac{\Delta \vdash M : T}{\Delta \vdash (\Lambda \alpha. M) : \forall \alpha. T} \quad (\alpha \notin \text{FTV}(\Delta))$	$\frac{\Delta \vdash M : \forall \alpha. T}{\Delta \vdash (M t) : T\{t/\alpha\}}$
$\frac{\Delta \vdash M : \text{nat} \quad \Delta \vdash N : \text{nat} \quad \Delta \vdash P : \text{nat}}{\Delta \vdash (\text{pifz } M N P) : \text{nat}}$	
$\frac{\Delta \vdash M : T \quad \Delta, x : T \vdash N : t}{\Delta \vdash (\text{let } x : T = M \text{ in } N) : t}$	

Table 6 Additional Reduction Rules for PPCF+XML.

$\frac{((\Lambda \alpha. M) s) \rightarrow M\{s/\alpha\} \quad (\text{let } x = N \text{ in } M) \rightarrow M\{N/x\}}{(\text{pifz } M N P) \rightarrow (\text{pifz } M' N P)}$	$\frac{(\text{pifz } 0 M N) \rightarrow M \quad (\text{pifz } (\text{succ } n) M N) \rightarrow N \quad (\text{pifz } M n n) \rightarrow n}{(M t) \rightarrow (M' t)}$
$\frac{M \rightarrow M' \quad N \rightarrow N'}{(\text{pifz } M N P) \rightarrow (\text{pifz } M' N' P)}$	
$\frac{P \rightarrow P'}{(\text{pifz } M N P) \rightarrow (\text{pifz } M N P')}$	

3 Translation from PCF to PPCF+XML

Pick a type variable α . Given a PCF type t we obtain a PPCF+XML type t^* by replacing each occurrence of `nat` by α . Assume that the type context of the term to be translated is $\Gamma = x_1 : t_1, \dots, x_n : t_n$. We build the translation in a few stages:

1. Define the type $\text{Cl}(\Gamma t) = t_1^* \rightarrow \dots \rightarrow t_n^* \rightarrow t^*$ (Cl here is for “closure”). If Γ is empty then this type is just t^* .
2. For a term M , let $M_{\Gamma t}^* = (\lambda x_1 : t_1 \dots \lambda x_n : t_n . M)$. Notice that this depends on the ordering of the $x_i : t_i$'s in Γ . Again, if Γ is empty then $M_{\Gamma t}^*$ is just M . If $\Gamma \vdash M : t$ is a derivable PCF typing judgement then clearly we have $\vdash M_{\Gamma t}^* : \text{Cl}(\Gamma t)\{\text{nat}/\alpha\}$.
3. The type scheme $\text{Con}(\Gamma t)$ (Con is for “context”) is

$$\forall \alpha. (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow (\alpha \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \text{Cl}(\Gamma t) \rightarrow \alpha$$

4. The translation of M , with respect to Γt , is the term $\llbracket M \rrbracket_{\Gamma t}$ given by

$$\llbracket M \rrbracket_{\Gamma t} = p \text{ nat succ pred } 0 \text{ ifz } M_{\Gamma t}^*$$

where $\text{succ} = (\lambda x : \text{nat} . \text{succ } x)$ and so on. In $\llbracket M \rrbracket$, p is any variable. If $\Gamma \vdash M : t$ is a derivable typing judgement in PCF, then

$$p : \text{Con}(\Gamma t) \vdash \llbracket M \rrbracket_{\Gamma t} : \text{nat}$$

is derivable in the polymorphic language.

The main result is that the translation $\llbracket \cdot \rrbracket$ preserves and reflects observational equivalence.

Theorem 4 Full Abstraction. *Suppose $\Gamma \vdash M : t$ and $\Gamma \vdash N : t$. Then*

$$\Gamma \vdash M \equiv N \iff p : \text{Con}(\Gamma t) \vdash \llbracket M \rrbracket_{\Gamma t} \equiv \llbracket N \rrbracket_{\Gamma t}$$

Proof. (Sketch) The (\Leftarrow) direction is not difficult. The main steps in the (\Rightarrow) direction are as follows.

1. Show that if $\llbracket M \rrbracket_{\Gamma t}$ and $\llbracket M' \rrbracket_{\Gamma t}$ are distinguishable, they are distinguishable in a context of the form (let $p : \text{Con}(\Gamma t) = P$ in $\llbracket \cdot \rrbracket$). This follows from a version of the Context Lemma [10].
2. Show that, for the purpose of making such distinctions, it is sufficient to consider P 's defined from the pure simply-typed λ -calculus involving only a divergent term Ω . The polymorphic type of p is essential for this.
3. Prove that for P as in 2, (let $p : \text{Con}(\Gamma t) = P$ in $\llbracket \llbracket M \rrbracket \rrbracket$) reduces to $C[M]$, where $C[\cdot]$ is a sequential PCF context, and similarly for N .

These properties allow us to construct a sequential PCF context that distinguishes M and N when $\llbracket M \rrbracket$ and $\llbracket N \rrbracket$ are inequivalent in PPCF+XML.

4 Translation from IA to PPCF+XML

The translation from IA to PPCF+XML goes in two stages. We define a denotational semantics-style encoding of IA into PCF called the concrete translation. We then give an abstract translation which uses the concrete translation.

4.1 The Concrete Translation

The concrete translation, on the level of types, goes as follows.

$$\begin{aligned}\mathcal{C}[\text{comm}] &= \text{nat} \rightarrow S \rightarrow S & L &= \text{nat} \\ \mathcal{C}[\text{exp}] &= \text{nat} \rightarrow S \rightarrow V & V &= \text{nat} \\ \mathcal{C}[\text{loc}] &= \text{nat} \rightarrow S \rightarrow L & S &= L \rightarrow V \\ \mathcal{C}[\theta \rightarrow \theta'] &= \mathcal{C}[\theta] \rightarrow \mathcal{C}[\theta']\end{aligned}$$

The extra **nat** parameter in the base types is used to keep track of the number of locations that have been allocated. The translation on terms is

$$\begin{aligned}\mathcal{C}[y] &= y \\ \mathcal{C}[0] &= \lambda x : \text{nat}. \lambda s : S. 0 \\ \mathcal{C}[\text{succ } M] &= \lambda x : \text{nat}. \lambda s : S. \text{succ}(\mathcal{C}[M] x s) \\ \mathcal{C}[\text{pred } M] &= \lambda x : \text{nat}. \lambda s : S. \text{pred}(\mathcal{C}[M] x s) \\ \mathcal{C}[\text{skip}] &= \lambda x : \text{nat}. \lambda s : S. s \\ \mathcal{C}[(M N)] &= (\mathcal{C}[M] \mathcal{C}[N]) \\ \mathcal{C}[\lambda z : \theta. M] &= \lambda z : \mathcal{C}[\theta]. \mathcal{C}[M] \\ \mathcal{C}[\mathbf{Y}_\theta M] &= \mathbf{Y}_{\mathcal{C}[\theta]} \mathcal{C}[M] \\ \mathcal{C}[C_1; C_2] &= \lambda x : \text{nat}. \lambda s : S. \mathcal{C}[C_2] x (\mathcal{C}[C_1] x s) \\ \mathcal{C}[M := E] &= \lambda x : \text{nat}. \lambda s : S. (s \mid (\mathcal{C}[M] x s) \mapsto (\mathcal{C}[N] x s)) \\ \mathcal{C}[\text{contents } M] &= \lambda x : \text{nat}. \lambda s : S. (s(\mathcal{C}[M] x s)) \\ \mathcal{C}[\text{ifz } M N P] &= \lambda x : \text{nat}. \lambda s : S. \text{ifz}(\mathcal{C}[M] x s)(\mathcal{C}[N] x s)(\mathcal{C}[P] x s) \\ \mathcal{C}[\text{new } z = E \text{ in } C] &= \\ &\quad \lambda x : \text{nat}. (\lambda z : \mathcal{C}[\text{loc}]. \lambda s : S. (((\mathcal{C}[C])(\text{succ } x) (s \mid (\text{succ } x) \mapsto (\mathcal{C}[E] x s)))) \\ &\quad \quad \quad \mid (\text{succ } x) \mapsto s(\text{succ } x))) \\ &\quad) (\lambda n : \text{nat}. \lambda s : S. (\text{succ } x))\end{aligned}$$

There are some obvious provisos here about free variables, e.g., in the equation for $\mathcal{C}[\text{succ } M]$, x cannot be free in M . The interpretation of ifz_{comm} requires a higher-order ifz in PCF, but this can be easily encoded. Also, in these definitions $(s \mid a \mapsto b)$ stands for

$$\lambda \ell : L. \text{ifz}(\text{eq } b \ b) (\text{ifz}(\text{eq } \ell \ a) \ b \ s(\ell)) \ \Omega$$

where $(\text{eq } c \ d)$ is itself sugar for an expression that returns 0 if c and d are defined and equal, 1 if they are defined and unequal, and diverges if either is undefined. The $(\text{eq } b \ b)$ test serves merely to make $(s \mid a \mapsto b)$ strict in b . It is not hard to see that the judgement $\pi \vdash M : \theta$ in IA gets translated to $\mathcal{C}[\pi] \vdash \mathcal{C}[M] : \mathcal{C}[\theta]$, where $\mathcal{C}[\pi]x = \mathcal{C}[\pi(x)]$ for $x \in \text{dom}(\pi)$.

Most of the valuations are self-explanatory except for **new**. Suppose we are evaluating $(\text{new } z = E \text{ in } C)$ in a state s where there are x active locations.

Then we evaluate the body C in a state where there are $x + 1$ locations, and where the extra location (which is itself simply $\text{succ } x$) has contents $C[E] \ x\ s$. This is the intuition behind $(C[C](\text{succ } x)(s \mid (\text{succ } x) \mapsto (C[E] \ x\ s)))$. The $\mid (\text{succ } x) \mapsto s(\text{succ } x)$ part restores $\text{succ } x$ to its old value on termination of the block. Finally, the argument that is passed to z simply serves to bind z to (an expression for) the new location.

The concrete translation yields a semantics that is very poor in many respects. For example, locations are represented as the type nat ; this has the disadvantage of there being an “undefined” location. Despite this extra baggage the concrete translation is still adequate.

Theorem 5 Adequacy. $C \Downarrow \iff (C[C] \ 0 \ (\lambda x : \text{nat}.x) \ 0) \Downarrow$.

Proof. (Sketch) If we compose the translation $C[\cdot]$ with the usual continuous function model of PCF, then we obtain a denotational semantics of IA. The adequacy of this denotational model for IA, together with the adequacy of the continuous function model for PCF, yields the result. The adequacy of this model of IA can be shown using standard methods (as in [7]). The only subtlety involves dealing with the “extra baggage,” such as non-definable commands in the semantics that are non-strict in their state argument; this requires some care when proving, using a computability argument as in [19], that operational termination implies semantic termination.

The three arguments to $C[C]$ in this result specify a context of evaluation where there are 0 initial locations and $(\lambda x : \text{nat}.x)$ is the initial state. The final argument is a location whose contents we look up to get a nat .

4.2 The Abstract Translation

We translate a judgement $\pi \vdash M : \theta$ in IA to a judgement in PPCF+XML, using the translation $C[\cdot]$ as an intermediary. Assume that $\pi = x_1 : \theta_1, \dots, x_n : \theta_n$, and let comm , exp and loc be distinct type variables. If θ is an IA type, let θ^* is the PPCF+XML type obtained by replacing occurrences of comm by comm , exp by exp , and loc by loc . The translation is defined in a few stages:

1. Define the type $\text{Cl}(\pi\theta) = \theta_1^* \rightarrow \dots \rightarrow \theta_n^* \rightarrow \theta^*$.
2. For a term M , let $M_{\pi\theta}^* = (\lambda x_1 : C[\theta_1], \dots \lambda x_n : C[\theta_n]. C[M])$. If π is empty, then $M_{\pi\theta}^*$ is just $C[M]$. If $\pi \vdash M : \theta$ is a derivable IA typing judgement,

$$\vdash M_{\pi\theta}^* : \text{Cl}(\pi\theta)\{\mathcal{C}[\text{comm}]/\text{comm}, \mathcal{C}[\text{exp}]/\text{exp}, \mathcal{C}[\text{loc}]/\text{loc}\}.$$

3. Define $\text{Con}(\pi\theta)$ to be

$$\begin{aligned} & \forall \text{comm. } \forall \text{exp. } \forall \text{loc. } (\text{exp} \rightarrow \text{exp}) \rightarrow (\text{exp} \rightarrow \text{exp}) \rightarrow \text{exp} \\ & \quad \rightarrow (\text{exp} \rightarrow \text{exp} \rightarrow \text{exp} \rightarrow \text{exp}) \rightarrow (\text{exp} \rightarrow \text{loc} \rightarrow \text{loc} \rightarrow \text{loc}) \\ & \quad \rightarrow (\text{exp} \rightarrow \text{comm} \rightarrow \text{comm} \rightarrow \text{comm}) \\ & \quad \rightarrow (\text{loc} \rightarrow \text{exp} \rightarrow \text{comm}) \rightarrow (\text{comm} \rightarrow \text{comm} \rightarrow \text{comm}) \\ & \quad \rightarrow (\text{loc} \rightarrow \text{exp}) \rightarrow (\text{exp} \rightarrow (\text{loc} \rightarrow \text{comm}) \rightarrow \text{comm}) \rightarrow \text{comm} \\ & \quad \rightarrow \text{Cl}(\pi\theta) \rightarrow \text{comm}. \end{aligned}$$

The types on the first line are for successor, predecessor and zero, those on the second and third lines are for conditionals, and those on the fourth and fifth lines are for assignment, sequencing, dereferencing, variable declarations, and skip.

4. The translation of M , with respect to $\pi\theta$, is the term $\mathcal{A}[M]_{\pi\theta}$ given by

$$\begin{aligned} \mathcal{A}[M]_{\pi\theta} = p \; \mathcal{C}[\text{comm}] \; \mathcal{C}[\text{exp}] \; \mathcal{C}[\text{loc}] \; \text{succ} \; \text{pred} \; \text{zero} \; \text{ifz}_{\text{exp}} \; \text{ifz}_{\text{loc}} \; \text{ifz}_{\text{comm}} \\ \text{assign} \; \text{seq} \; \text{contents} \; \text{new} \; \text{skip} \; M^*_{\pi\theta} \\ 0 \; (\lambda x : \text{nat}.x) \; 0 \end{aligned}$$

where $\text{succ} = \mathcal{C}[\lambda x : \text{exp}. \; \text{succ } x]$, and so on. In $\mathcal{A}[M]$, p is some “fresh” variable. If $\pi \vdash M : \theta$ is a derivable typing judgement in IA, then

$$p : \text{Con}(\pi\theta) \vdash \mathcal{A}[M]_{\pi\theta} : \text{nat}$$

is derivable in the polymorphic language.

The term $\mathcal{A}[M]$, minus its last three arguments, is of type $\mathcal{C}[\text{comm}]$. The final three arguments play the same role as in the statement of the adequacy of the concrete translation.

Theorem 6 Full Abstraction. *Suppose $\pi \vdash M : \theta$ and $\pi \vdash N : \theta$. Then*

$$\pi \vdash M \equiv N \iff p : \text{Con}(\pi\theta) \vdash \mathcal{A}[M] \equiv \mathcal{A}[N]$$

Proof. The proof runs along the same lines as for Theorem 4, with the exception that step 3 needs to be modified as follows.

3. ($\text{let } p : \text{Con}(\pi\theta) = P \text{ in } [\mathcal{A}[M]]$) reduces to $C[\mathcal{C}[M]]$, where $C[\cdot]$ is in the image of $\mathcal{C}[\cdot]$, and similarly for N .

The adequacy of $\mathcal{C}[\cdot]$ is then used to build a distinguishing context.

5 Conclusion

We have given fully abstract translations from sequential PCF and an idealized Algol into a parallel extension of PCF with ML-style polymorphism. Our translation method appears to be fairly general, and it would be interesting to attempt to apply it to other languages. One particularly important area concerns dynamic allocation [18], or a combination of dynamic allocation and reference types like that found in Standard ML. We expect that a stronger type theory than XML—probably involving the addition of recursive types—would be needed to treat references that store stateful objects, such as other references. Our translation techniques also might be applicable to other languages, such as a language with concurrency (e.g., Reppy’s Concurrent ML [21]), a language with exceptions, or a language with continuations. While this discussion is speculative, more examples would lend support to the idea of parametricity as a unifying concept, a proper understanding of which may shed light on diverse problems in programming theory.

Proving that a denotational model of a programming language is fully abstract boils down to showing that the “abstraction barrier” between the abstract description of a programming language and a denotational semantics “implementation” is maintained. For instance, the standard continuous function model of PCF is *not* fully abstract because the model contains certain “parallel” functions that can be used to distinguish terms that are not distinguishable in the sequential setting [19, 26]. Similarly, standard models of Algol are not fully abstract [9]: there are functions in these model that violate the “stack discipline” of local variables and that can be used to distinguish observationally indistinguishable terms. In these models, a “denotational program” is allowed access to operations that are not provided directly by the language. Our translations into a PPCF+XML show that parametricity may be used to prevent such unauthorized access, and that a fully abstract model of PPCF+XML would provide a basis for reasoning about PCF and IA.

Nevertheless, there are two potential limitations to our results. Firstly, our translations probably do not yield “practical” methods for reasoning about PCF or IA. Reasoning about a PCF or IA term directly from the translation would involve reasoning about a large polymorphic application. Given a suitable model of PPCF+XML, it would be preferable to bypass the application of a polymorphic function and consider directly the meanings determined by such an application. One might expect that a more direct characterization of such meanings might be possible with a little work, but this is difficult to predict in the absence of a fully abstract model of PPCF+XML.

Secondly, our results do not definitively establish that a solution to the full abstraction problem for PPCF+XML entails a solution to the problems for PCF and IA. One difficulty, of course, is that the word “solution” here is ill-defined. In the case of PCF, one would at least want a characterization of Milner’s unique model [10] (which we believe would be possible) together with a characterization of finite elements and some “semantic” conditions explaining the “sequential” nature of the function type (which is not immediate from our translation). In the case of IA, one would perhaps like to see structure, such as that in functor category models [23, 15, 13], explaining the *variance* in the concept of state that is caused by variable declarations. Again, though we believe that there may be reasonable possibilities in this direction, without a fully abstract model of PPCF+XML it is not possible to predict with assurance that “good” models of PCF or IA would be obtained.

In this paper we purposely restricted our attention to ML-style polymorphism in order to emphasize the point that issues of parametricity and full abstraction are interesting even for this limited variety of polymorphism. Indeed, most studies of parametricity have taken place in the context of the second-order polymorphic λ -calculus (*e.g.* [2, 4, 8, 24]). ML’s polymorphism is of a comparatively simple form, where type variables never themselves get instantiated to polymorphic types. This *predicative* flavour would allow parametricity to be examined in isolation from the foundational issues raised by the impredicative polymorphism of the second-order λ -calculus. Thus, ML’s polymorphism might

serve as a useful test bed for examining rigorous explanations of parametricity, as an intermediate step on the way to a more general understanding.

ML-style polymorphic types can be interpreted quite straightforwardly as indexed products of domains, trimmed by Reynolds's relational-parametricity conditions [24]; using the Kripke logical relations of [6] may work especially well. There are also a number of PER models that possess appropriate domain-theoretic structure (e.g. [3, 17]). Some of these models perform quite well at certain "low-order" types, but little is known at higher types, and it is not known if any of them is fully abstract. The presence of `pif` in PPCF+XML may help in providing such a semantic model (cf. [19]), e.g., in considering definability results for polymorphic types with embedded occurrences of `nat`. Of course, full abstraction for the polymorphic language without `pif` (not so far-fetched a possibility in light of some recent preliminary announcements [1, 16]) would also be interesting in connection to the results presented here for the purpose of studying Algol (our translation result for IA does not *require pif*).

One aspect of state that we have not emphasized—but which is at least as fundamental as local variables—is what is often called the "single threaded" nature of state: when a state change occurs, the old state disappears. Even the most advanced models of Algol—e.g., the ones described in [14, 27]—are known not to be fully abstract without what Reynolds calls "snap-back" operations. These operations violate the single-threaded nature of Algol by allowing backtracking of state changes during evaluation, requiring the maintenance of a stack of (temporary) states instead of only one. It is interesting, then, that our translation results do not require the presence of snap-back operations. This is very similar to the connection between single-threading and abstract types suggested by Wadler [29]. Wadler's observations were made in the context of functional programming, where single-threadedness is needed for the safety of in-place array update, but they also appear to be relevant to the understanding of semantic aspects of single-threading. Reddy [20] has developed a different approach to single-threading by applying and extending some ideas from linear logic.

An important precursor to this work is the paper of O'Hearn and Tennent [14], where a connection between block structure and parametricity was first proposed (a related work is [27]). The information-hiding that is obtained through encapsulating pieces of local state was explained there using a denotational model that borrowed ideas from Reynolds's relational approach to parametricity. They obtain good characterizations of the structure of "low order" Algol types (related to initial algebra results for polymorphism, e.g. [24, 4]), but full abstraction issues have not been resolved for their models. In contrast, we obtain a (syntactic) full abstraction result, but not a direct characterization of "low order" types. Another difference is that our translation only requires ML-style polymorphism, whereas their semantics contains parametric functions as arguments to other functions. While neither our translation nor our results are directly analogous to [14], we consider our results as providing further evidence in favour of a connection between local state and parametricity.

References

1. S. Abramsky, R. Jagadeesan, and P. Malacaria. Games and full abstraction for PCF: preliminary announcement. Unpublished, 1993.
2. S. Bainbridge, P. J. Freyd, A. Scedrov, and P. J. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70(10):35–64, 1990. *Corrigendum* in 71(3):431, 1990.
3. P. J. Freyd, P. Mulry, G. Rosolini, and D. S. Scott. Extensional PERs. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 346–354, Philadelphia, PA, 1990. IEEE Computer Society Press, Los Alamitos, California.
4. P. J. Freyd, E. P. Robinson, and G. Rosolini. Functorial parametricity. In *Proceedings, 7th Annual IEEE Symposium on Logic in Computer Science*, pages 444–452, Santa Cruz, California, 1992. IEEE Computer Society Press, Los Alamitos, California.
5. R. Harper and J. C. Mitchell. On the type structure of standard ML. *ACM Trans. Programming Languages and Systems*, 15:211–252, 1993.
6. A. Jung and J. Tiuryn. A new characterization of lambda definability. In *Typed Lambda Calculi and Applications*, volume 664 of *Lect. Notes in Computer Sci.*, pages 245–257. Springer-Verlag, 1993.
7. A. F. Lent. The category of functors from state shapes to bottomless CPOs is adequate for block structure. Master’s thesis, Massachusetts Institute of Technology, 1992.
8. Q. Ma and J. C. Reynolds. Types, abstraction, and parametric polymorphism, part 2. In S. Brookes et al., editors, *Mathematical Foundations of Programming Semantics*, volume 598 of *Lecture Notes in Computer Science*, pages 1–40. Springer-Verlag, Berlin, 1992. Proceedings of the 1991 Conference.
9. A. R. Meyer and K. Sieber. Towards fully abstract semantics for local variables: preliminary report. In *Conf. Record 15th ACM Symp. on Principles of Programming Languages*, pages 191–203. ACM, New York, 1988.
10. R. Milner. Fully abstract models of typed λ -calculi. *Theoretical Computer Science*, 4:1–22, 1977.
11. R. Milner. A theory of type polymorphism in programming. *J. of Computer and System Sciences*, 17:348–75, 1978.
12. J. C. Mitchell and R. Harper. The essence of ML. In *Conf. Record 15th ACM Symp. on Principles of Programming Languages*, pages 28–46. ACM, New York, 1988.
13. P. W. O’Hearn and R. D. Tennent. Semantics of local variables. In M. P. Fourman, P. T. Johnstone, and A. M. Pitts, editors, *Applications of Categories in Computer Science*, volume 177 of *London Mathematical Society Lecture Note Series*, pages 217–238. Cambridge University Press, Cambridge, England, 1992.
14. P. W. O’Hearn and R. D. Tennent. Parametricity and local variables. Technical Report SU-CIS-93-30, Syracuse University, 1993. Preliminary version appeared in *Conf. Record 20th ACM Symp. on Principles of Programming Languages*, pages 171–184, Charleston, South Carolina, 1993. ACM, New York.
15. F. J. Oles. *A Category-Theoretic Approach to the Semantics of Programming Languages*. Ph.D. thesis, Syracuse University, Syracuse, N.Y., 1982.
16. L. Ong and M. Hyland. Dialogue games and innocent strategies: An approach to intensional full abstraction to PCF (preliminary announcement). Unpublished, 1993.

17. W. K. Phoa. Effective domains and intrinsic structure. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 366–379, Philadelphia, PA, 1990. IEEE Computer Society Press, Los Alamitos, California.
18. A. Pitts and I. Stark. On the observable properties of higher-order functions that dynamically create local names (preliminary report). In *ACM SIGPLAN Workshop on State in Programming Languages*, pages 31–45, 1993. Available as Yale Technical Report YALEU/DCS/RR-968.
19. G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.
20. U.S. Reddy. Global states considered unnecessary. In *ACM SIGPLAN Workshop on State in Programming Languages*, 1993. Available as Yale Technical Report YALEU/DCS/RR-968.
21. J. H. Reppy. CML: A higher-order concurrent language. In *Proceedings of the SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 293–305, June 1991.
22. J. C. Reynolds. Towards a theory of type structure. In *Proc. Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425, Berlin, 1974. Springer-Verlag.
23. J. C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372. North-Holland, Amsterdam, 1981.
24. J. C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523. North Holland, Amsterdam, 1983.
25. J. G. Riecke. Fully abstract translations between functional languages. *Mathematical Structures in Computer Science*, 1993. To appear.
26. V. Sazonov. Expressibility of functions in D. Scott's LCF language. *Algebra i Logika*, 15:308–330, 1976. Russian.
27. K. Sieber. New steps towards full abstraction for local variables. In *ACM SIGPLAN Workshop on State in Programming Languages*, pages 88–100, 1993. Available as Yale Technical Report YALEU/DCS/RR-968.
28. C. Strachey. *Fundamental Concepts in Programming Languages*. Unpublished lecture notes, International Summer School in Computer Programming, Copenhagen, August 1967.
29. P. Wadler. Comprehending monads. In *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 61–78, Nice, 1990.

BROADCASTING WITH PRIORITY

K. V. S. PRASAD

ABSTRACT. Adding priorities to CCS is difficult, and involves two-stage operational semantics or other complications. By contrast, priorities can be added very simply to a calculus of broadcasting systems (CBS). The reason is the input/output distinction made in CBS, with output actions being autonomous. Priority makes sense only for autonomous actions.

As in unprioritised CBS, both strong and weak bisimulation are congruences, and capture the intuitively desired equivalences. CBS is also a powerful and natural language, offering an interesting new programming paradigm. Several examples show that priorities extend both the power and the paradigm.

The public address system at an airport can give bomb alerts, say, priority over flight announcements. Everyone hears messages read out over the system, but individual responses vary. Priority assignment cannot depend on the actual public responses to the contending messages, even if expectations about responses play a role. Priority here is only a matter of choosing between the messages themselves.

Priorities in the marketplace are more complex, because buyer and seller must agree for a sale to take place. Sellers' minimum prices and buyers' maximum prices must be met, but further preferences for advantageous prices are not relevant unless there are competing parties to do business with. A broker arranging sales that respect everybody's priorities has a harder task than the announcer at the airport.

The handshake (or rendezvous) model of communication is similar to the market model. An indication of the difficulty of respecting priorities in this model is the "priority inversion" problem in Ada [BA90]. This model is also predominant in process calculus. The several papers [BBK85, CH88, CW91] that add priority to process calculus have all used this model, and therefore suffer from the broker's difficulties. In [CH88], an a priori semantics works out what might happen, and a second stage works out what actually happens. In [CW91], transitions are predicated on what the environment can do.

This paper adds priorities to a calculus of broadcasting systems (CBS) [Pra93c]. The resulting CBS with priorities (PCBS) is strikingly simple, and compares very favourably with CCS with priorities. This confirms the intuition above, since the

Key words and phrases. Broadcast, priority, process calculi, bisimulation, functional programming.

Author's address: Department of Computing Science, Chalmers University of Technology, S-412 96 Gothenburg, Sweden. *E-mail:* prasad@cs.chalmers.se.

Funding: From the Swedish Government agencies TFR and NUTEK, the latter supporting Chalmers' membership of the Esprit Basic Research Action CONCUR2.

communication model of CBS resembles a public address system. It also demonstrates that there is no inherent difficulty in incorporating priorities into the mathematical apparatus of operational semantics and observational equivalences.

The ease of the transition from CBS to PCBS might lead one to doubt that anything real has been achieved. Several programming examples follow to quell such doubts. They show novel uses of priority far beyond simple interrupts. PCBS, like CBS, is a programming language as well as a process calculus. A simulator for PCBS has been implemented, and all the examples here have been run on it.

Readers are assumed to be familiar with CCS [Mil89] and with functional programming. Familiarity with [Pra93c] is not strictly necessary, but would help. More discussion of the CBS model of computation can be found in [Pra93b, Pra91], which present older versions of CBS, significantly different from that of [Pra93c].

1. THE SYNTAX AND SEMANTICS OF PCBS

A PCBS process of type $\text{Proc } \alpha$ says or hears values of type α and evolves to other processes of type $\text{Proc } \alpha$. Saying and hearing $v:\alpha$ are written $\xrightarrow{v;k}$ and $\xrightarrow{v?k}$, where k is the priority of the action. Priorities are natural numbers, with 0 the highest. CBS is the special case of PCBS where k is always 0.

A useful auxiliary notion is that of the priority of a process, defined to be k if it has an utterance at priority k . There can be only one such k ; if a process has several components, only the most urgent will speak. A process with nothing to say is said to have priority “ ∞ ”.

In (P)CBS, whatever is said is heard by all processes in parallel with the speaker. In PCBS, hearing is read as encoded permission to speak: a process with things to say will not hear less urgent speech, which therefore cannot take place.

The syntax and semantics of PCBS are given in Table 1. Priorities are used only in the rules for translation and in the side conditions for guarded sums. The calculus abstracts away from the details of the evaluation, written \Downarrow , of data expressions. The data type α may itself involve the type $\text{Proc } \beta$ for some β . This paper does not explore this possibility, i.e., attention is restricted to first order PCBS.

The guarded sum $\sum_{i \in I} e_i!_k, P_i + ?f$ is distfix notation for a single constructor with several arguments, e_i , k_i , P_i and f . NOTE: $e!_k P$ is not a process by itself, and there is no general binary $+$. The special cases $?f$, where $I = \emptyset$, and $e!_k P + ?f$, where I is a singleton, are the only ones used in programming.

The process $?f$ has nothing to say. It hears any utterance v of the environment, no matter at what priority it is said, and becomes the process $f v$.

$$?f \xrightarrow{5?3} f 5 \quad ?\lambda x. ((x + 1)!_3 P + ?f) \xrightarrow{5?2} (5 + 1)!_3 P + ?f$$

\circ can be defined directly or derived upto \sim (strong bisimulation, see Section 4).

$$\circ \xrightarrow{v?k} o \quad o \sim X \text{ where } X \stackrel{\text{def}}{=} ?\lambda x. X$$

Only closed processes communicate, not abstractions like $\lambda x. (x + 1)!_3 P$ or expressions with free data variables like $(x + 1)!_3 P$. The process $(5 + 1)!_3 P + ?f$ has priority 3; it wishes to say 6 at priority 3 and become P . If it hears v , which has to be said at priority 3 or higher, it evolves to $f v$.

$$(5 + 1)!_3 P + ?f \xrightarrow{6!3} P \quad (5 + 1)!_3 P + ?f \xrightarrow{4?k} f 4 \text{ if } k \leq 3$$

Let α be a datatype. Let τ be a distinguished value, and α_τ the disjoint union of α and $\{\tau\}$. Let $P, P;: Proc \alpha$, the type of processes communicating α 's. Let I be a finite set. Let $e_i:\alpha_\tau$, $k_i:\mathbb{N}$ and $f:\alpha \rightarrow Proc \alpha$.

Let $P_\beta: Proc \beta$. Let $\phi = \langle k_\phi, \phi^\dagger, \phi_1 \rangle$, where $k_\phi:\mathbb{N}$, $\phi^\dagger:\beta_\tau \rightarrow \alpha_\tau$ and $\phi_1:\alpha_\tau \rightarrow \beta_\tau$ satisfying $\phi^\dagger\tau = \tau$ and $\phi_1\tau = \tau$.

Let $b: Bool$, and let A range over constants, declared in (mutually) recursive guarded definitions $A \stackrel{\text{def}}{=} P$. Then the elements of $Proc \alpha$ are given by

$$P ::= \sum_{i \in I} e_i!_{k_i} P_i + ?f \mid P \parallel P \mid \phi P_\beta \mid \text{if } b \text{ then } P \text{ else } P \mid A$$

$v:\alpha$, $w:\alpha_\tau$ and $u:\beta_\tau$ are canonical values. The relation \Downarrow means “can evaluate to”. $\xrightarrow{w!k}$ and $\xrightarrow{w?k}$ are actions at priority k , where $k:\mathbb{N}$. Let $P', P'_i: Proc \alpha$; and $P'_\beta: Proc \beta$.

Guarded	$\sum_{i \in I} e_i!_{k_i} P_i + ?f \xrightarrow{w!k_j} P_j \quad j \in I, e_j \Downarrow w, \text{ and } k_j = \min_{i \in I} k_i$									
Sum	$\sum_{i \in I} e_i!_{k_i} P_i + ?f \xrightarrow{\tau?k} \sum_{i \in I} e_i!_{k_i} P_i + ?f \quad k \leq \min_{i \in I} k_i$									
	$\sum_{i \in I} e_i!_{k_i} P_i + ?f \xrightarrow{v?k} f v \quad k \leq \min_{i \in I} k_i$									
Compose ¹	$\frac{P_1 \xrightarrow{w!k_1} P'_1 \quad P_2 \xrightarrow{w!k_2} P'_2 \quad k_1 * k_2 \neq \nabla}{P_1 \parallel P_2 \xrightarrow{w!(k_1 * k_2)} P'_1 \parallel P'_2}$ <table border="1" style="float: right; margin-left: 10px;"> <tr> <td>•</td><td>!</td><td>?</td></tr> <tr> <td>!</td><td>▀</td><td>!</td></tr> <tr> <td>?</td><td>!</td><td>?</td></tr> </table>	•	!	?	!	▀	!	?	!	?
•	!	?								
!	▀	!								
?	!	?								
Translate	$\frac{P_\beta \xrightarrow{u!k} P'_\beta}{\phi P_\beta \xrightarrow{\phi^\dagger u!(k+k_\phi)} \phi P'_\beta} \quad \frac{P_\beta \xrightarrow{\phi_1 w?(k-k_\phi)} P'_\beta}{\phi P_\beta \xrightarrow{w?k} \phi P'_\beta}$									
Conditional ¹	$\frac{P_1 \xrightarrow{w!k} P'_1 \quad \text{if } b \text{ then } P_1 \text{ else } P_2 \xrightarrow{w!k} P'_1}{b \Downarrow \text{true}} \quad \frac{P_2 \xrightarrow{w!k} P'_2 \quad \text{if } b \text{ then } P_1 \text{ else } P_2 \xrightarrow{w!k} P'_2}{b \Downarrow \text{false}}$									
Define ¹ $A \stackrel{\text{def}}{=} P$	$\frac{P \xrightarrow{w!k} P'}{A \xrightarrow{w!k} P'}$									

¹ $\natural, \natural_1, \natural_2$ range over $\{!, ?\}$. ∇ means “undefined” in the synchronisation algebra for \bullet .

TABLE 1. The syntax of PCBS and the semantics of closed processes

A special case of $e!_k P + ?f$ is important. Let

$$e!!_k P \stackrel{\text{def}}{=} e!_k P + ?\lambda x. (e!!_k P)$$

Abusing notation, $e!!_k P$ is written $e!_k P$. Thus “ $e!_k P$ ” outside a $+ ?$ context stands for “ $e!!_k P$ ”. Suppose $e \Downarrow v$, i.e., e evaluates to v . The process $e!_3 P$ wishes to say v at priority 3; it hears, but ignores, anything at priority 3 or lower.

$$e!_k P \xrightarrow{v!k} P \text{ if } e \Downarrow v \quad e!_k P \xrightarrow{v?k'} e!_k P \text{ if } k' \leq k$$

If $e!_k P$ is defined by these rules and $e!!_k P$ by recursion, then $e!_k P \sim e!!_k P$.

Communication is synchronous. At every step, one of the processes in a parallel composition says something, and all the other processes hear it. Speakers go one

at a time, and are independent of the number of listeners. $|$ is associative.

$$\begin{aligned} ?f \mid 5!_3 P \mid ?\lambda y. y!_1 o \xrightarrow{5!3} f \mid & P \mid 5!_1 o \\ ?f \mid 5!_3 P \mid ?\lambda y. y!_1 o \xrightarrow{4?2} f \mid & 4 \mid 5!_3 P \mid 4!_1 o \end{aligned}$$

All processes including parallel compositions respond deterministically to what they hear. The only guarded sums used in programming, $?f$ and $v!_k P + ?f$, are deterministic also for speech. Contending speakers in parallel are in fact the only source of non-determinism. To permit an expansion theorem, a more general guarded sum of the form $\sum_{i \in I} e_i !_{k_i} P_i + ?f$ is needed. Only finite parallel compositions are used, so I can be restricted to be a finite set.

$$\begin{aligned} 2!_1 P \mid 7!_2 Q \sim & 2!_1 (P \mid 7!_2 Q) + 7!_2 (2!_1 P \mid Q) + ?\lambda x. (2!_1 P \mid 7!_2 Q) \\ \sim & 2!_1 (P \mid 7!_2 Q) + ?\lambda x. (2!_1 P \mid 7!_2 Q) \end{aligned}$$

The second line follows because outranked output guards can never be used. The process $\sum_{i \in I} e_i !_{k_i} P_i + ?f$ has priority $\min_{i \in I} k_i$. It non-deterministically says one of the most urgent things it has to say, except if preempted by the environment.

$$\begin{aligned} \sum_{i \in I} e_i !_{k_i} P_i + ?f \xrightarrow{5?k} f \mid 5 & k \leq \min_{i \in I} k_i \\ \sum_{i \in I} e_i !_{k_i} P_i + ?f \xrightarrow{v!k_j} P_j & j \in I, e_j \Downarrow v, \text{ and } k_j = \min_{i \in I} k_i \end{aligned}$$

Finally, a translator ϕ is a triple $\langle k_\phi, \phi^\dagger, \phi_\downarrow \rangle$. The process ϕP says $\phi^\dagger 5$ at priority $k + k_\phi$ if P says 5 at k , and hears 4 at k if P hears $\phi_\downarrow 4$ at $k - k_\phi$. Here $n \dashv m$ is $n - m$ if $n \geq m$ and 0 otherwise. Thus a translator can deprioritise a subsystem.

Translators interface between systems of different types. Hiding and restriction are achieved by translation to τ , an aside appended to every data type. Asides can be spoken but not heard. In implementations of PCBS, the premise $P \xrightarrow{\tau!k} P'$ yields the conclusion $P \mid Q \xrightarrow{\tau!k} P' \mid Q$ provided $k \leq \pi(Q)$. This is equivalent to the more concise formulation here, where asides are heard but always ignored:

$$Q \xrightarrow{\tau?k} Q' \text{ where } Q' \equiv Q \text{ up to unfolding of constants and conditionals}$$

Translating functions must map τ to τ .

2. PROPERTIES OF THE TRANSITION SYSTEM

DEFINITION 1. The priority $\pi(P)$ of a closed process P is given by the rules

$$\begin{aligned} \pi(?f) &= \infty \\ \pi(\sum_{i \in I} e_i !_{k_i} P_i + ?f) &= \min_{i \in I} k_i \text{ if } I \neq \emptyset \\ \pi(P \mid Q) &= \min(\pi(P), \pi(Q)) \\ \pi(\langle k_\phi, \phi^\dagger, \phi_\downarrow \rangle P) &= k_\phi + \pi(P) \\ \pi(\text{if } b \text{ then } P_1 \text{ else } P_2) &= \text{if } b \text{ then } \pi(P_1) \text{ else } \pi(P_2) \\ \pi(A) &= \pi(F) \quad \text{where } A \stackrel{\text{def}}{=} F \end{aligned}$$

□

Let $P \xrightarrow{w!k}$ abbreviate “there exists P' such that $P \xrightarrow{w!k} P'$ ”, and let $P \xrightarrow{w!k \nexists}$ abbreviate “there is no P' such that $P \xrightarrow{w!k} P'$ ”.

PROPOSITION 2. $\pi(P) = k$ and $k \neq \infty$ iff $\exists v, P'$ such that $P \xrightarrow{v!k} P'$. Also, $P \xrightarrow{v!k} \nexists$ and $P \xrightarrow{v'!k'} \nexists$ implies $k = k'$. So $P \xrightarrow{w!k \nexists}$ if $k < \pi(P)$. □

REMARK 3. For all P , the set $\{(w, k) \mid P \xrightarrow{w!k}\}$ is finite. If priorities were to increase from 0 upwards, this fact would be needed to ensure that priority was well defined, for a process like $\sum_{i \in \mathbb{N}} e_i !_i P_i + ?f$ would stop all progress in any system of which it is a component. (A “priority stop” process). \square

PROPOSITION 4. $P \xrightarrow{w?k}$ iff $k \leq \pi(P)$. \square

PROPOSITION 5. $\forall P, w, \exists! P'$ such that $P \xrightarrow{w?0} P'$. \square

DEFINITION 6. P/w , the image of P under w , is the P' such that $P \xrightarrow{w?0} P'$. \square

PROPOSITION 7. $k \leq \pi(P)$ implies $P \xrightarrow{\tau?k} P'$, where $P \sim P'$. \square

In fact, if $k \leq \pi(P)$ then $P \xrightarrow{\tau?k} P'$, where $P \equiv P'$ up to unfolding of constants and conditionals. So $(?f)/v = f v$ and $P/\tau \sim P$. Section 4 defines \sim .

3. FROM CBS TO PCBS

It is instructive to consider alternative designs for PCBS, starting from the same syntax, and the same interpretation: $3!_1 A$ wishes to say “3” at priority 1. A first attempt could start from Definition 1, which is purely syntactic. The parallel rule could then be taken to be $P \xrightarrow{v!} P'$ and $Q \xrightarrow{v?} Q'$ and $\pi(P) \leq \pi(Q)$ imply $P | Q \xrightarrow{v!} P' | Q'$. It allows $3!_1 A | 4!_2 B \xrightarrow{3!} A | 4!_2 B$ to be derived, but not $3!_1 A | 4!_2 B \xrightarrow{4!} 3!_1 A | B$.

But in this calculus $P \sim R$ does not imply $P | Q \sim R | Q$. For $3!_1 A \sim 3!_2 A$, yet $3!_1 A | 5!_1 C$ can say “3”, while $3!_2 A | 5!_1 C$ cannot. To repair this, annotate speech with the speaker’s priority: $3!_1 A \xrightarrow{3!1} A$. Now $3!_1 A \not\sim 3!_2 A$. The annotation is only to help reasoning; it is not part of what is said. The announcer reads out the chosen message, not (usually) why it was chosen over others.

Proposition 2 holds for this intermediate calculus. The side-condition $\pi(P) \leq \pi(Q)$ is equivalent to $Q \xrightarrow{w!k}$ if $k < \pi(P)$. Despite this negative premise, the transition system is well defined, for the definition of $\pi(P)$ is independent of the transitions of P .

The final step to PCBS, that of annotating hearing with the priority of the speech heard, is less important than the annotation of speech. It encodes process priority into hearing transitions, and therefore absorbs the side-condition of the parallel rule into the second premise. Proposition 4 describes the new annotation.

4. STRONG BISIMULATION

Let \mathbb{P}_{cl} be the set of closed processes. Let $\natural \in \{!, ?\}$.

DEFINITION 8 (STRONG BISIMULATION FOR CLOSED PROCESSES). $\mathcal{R} \subseteq \mathbb{P}_{cl} \times \mathbb{P}_{cl}$ is a strong bisimulation if whenever $P \mathcal{R} Q$,

- (i) if $P \xrightarrow{w!k} P'$ then $\exists Q'$ such that $Q \xrightarrow{w!k} Q'$ and $P' \mathcal{R} Q'$,
- (ii) if $Q \xrightarrow{w!k} Q'$ then $\exists P'$ such that $P \xrightarrow{w!k} P'$ and $P' \mathcal{R} Q'$ \square

The largest strong bisimulation is an equivalence, denoted \sim . It can be extended to process abstractions thus: $f \sim g$ iff $\forall v, f v \sim g v$. So $f \sim g$ implies $?f \sim ?g$.

PROPOSITION 9. \sim is a congruence for CBS. \square

DEFINITION 10. For any $\mathbb{L} \subseteq \alpha$, if the transmissions $w!k$ of P and all its derivatives are such that $w \in \mathbb{L} \cup \{\tau\}$ then P has sort \mathbb{L} , written $P : \mathbb{L}$. \square

In $\sum_{i \in I} e_i !_{k_i} P_i + ?f$, the set I is finite. If we also assume a standard order for its elements, the “sum” above is just a list. Laws 1(a) and 1(b) below say that this list is a set, and justify the sum notation. Laws 1(c), 3(a), 3(c) and 4 below distinguish prioritised CBS from ordinary CBS. In Law 2, \mathbb{P} is the set of all processes.

PROPOSITION 11 (STRONG BISIMULATION LAWS).

- (1) (a) $\dots + e_i !_{k_i} P_i + e_j !_{k_j} P_j + \dots + ?f \sim \dots + e_j !_{k_j} P_j + e_i !_{k_i} P_i + \dots + ?f$
 (b) $\dots + e_i !_{k_i} P_i + e_i !_{k_i} P_i + \dots + ?f \sim \dots + e_i !_{k_i} P_i + \dots + ?f$
 (c) $\dots + e_i !_{k_i} P_i + e_j !_{k_j} P_j + \dots + ?f \sim \dots + e_i !_{k_i} P_i + \dots + ?f$ if $k_i < k_j$
- (2) $(\mathbb{P}/ \sim, |, \circ)$ is a commutative monoid.
- (3) (a) $\phi(e!_k P) \sim (\phi^\dagger e) !_{k+k_\phi} \phi P$
 (b) $\phi(?f) \sim X$ where $X \stackrel{\text{def}}{=} ?\lambda x. \text{if } \phi \downarrow x = \tau \text{ then } X \text{ else } \phi(f(\phi \downarrow x))$
 (c) $\phi(\sum_{i \in I} e_i !_{k_i} P_i + ?f) \sim X$ where
 $X \stackrel{\text{def}}{=} \sum_{i \in I} (\phi^\dagger e_i) !_{k_i+k_\phi} (\phi P_i) + ?\lambda x. \text{if } \phi \downarrow x = \tau \text{ then } X \text{ else } \phi(f(\phi \downarrow x))$
- (4) $\phi\psi P \sim \phi \circ \psi P$ where $\phi \circ \psi = \langle k_\phi + k_\psi, \phi^\dagger \circ \psi^\dagger, \psi \downarrow \circ \phi \downarrow \rangle$.
- (5) $\phi(P_1 | P_2) \sim \phi P_1 | \phi P_2$ if $\forall v \in \mathbb{L}_1 \cup \mathbb{L}_2, \phi \downarrow v = v$, where $P_1 : \mathbb{L}_1$ and $P_2 : \mathbb{L}_2$

\square

The expansion theorem below does not need to take care of priorities. Law 1(c) applied to the r.h.s. shows that all is well.

PROPOSITION 12 (EXPANSION THEOREM). Let $r \in \{0, 1\}$. Then

$$P_0 | P_1 \sim \sum_{r,w,k} w!_k (P'_r | P_{1-r}/w) + ?\lambda x. (P_0/x | P_1/x) \text{ where } P_r \xrightarrow{w!k} P'_r. \square$$

CBS⁺, the calculus of [Pra93b], is an unprioritised calculus with ? and ! prefixes, \circ and + as primitive constructors. The equations 1(a) and 1(b) of Proposition 11 together with the expansion theorem constitute a complete axiom system for finite processes of CBS⁺. This result has not yet been adapted to the CBS of [Pra93c]. PCBS was earlier built on CBS⁺, and it was then clear that 1(c) was the only law that distinguished the two. It seems a reasonable conjecture that this is the case in the present calculus as well.

DEFINITION 13 (PRIORITY ABSTRACTED BISIMULATION FOR CLOSED PROCESSES).

$\mathcal{R} \subseteq \mathbb{P}_{cl} \times \mathbb{P}_{cl}$ is a priority abstracted bisimulation if whenever $P \mathcal{R} Q$,

- (i) if $P \xrightarrow{w!k} P'$ then $\exists Q', k'$ such that $Q \xrightarrow{w!k'} Q'$ and $P' \mathcal{R} Q'$,
- (ii) if $Q \xrightarrow{w!k} Q'$ then $\exists P', k'$ such that $P \xrightarrow{w!k'} P'$ and $P' \mathcal{R} Q'$ \square

The largest priority abstracted bisimulation is an equivalence, denoted \simeq . This is not a congruence, as Section 4 pointed out, for $3!_1 A \simeq 3!_2 A$, yet $3!_1 A | 5!_1 C \xrightarrow{3!1} \tau$, while $3!_2 A | 5!_1 C \not\xrightarrow{3!1}$ for any k . Let \simeq^c be the largest congruence contained in \simeq .

PROPOSITION 14. $\sim = \simeq^c$

Proof. It is easy to see that $\sim \subseteq \simeq$, and \sim is a congruence. For the other direction, let $P \simeq^c Q$, and let $P \xrightarrow{w!k} P'$. Then $Q \xrightarrow{w!k'} Q'$, and $P' \simeq^c Q'$. But $k' = k$, otherwise a | context can be found that can distinguish P and Q . \square

5. WEAK BISIMULATION

For convenience, let $P \xrightarrow{\tau!} P'$ stand for $\exists k$ such that $P \xrightarrow{\tau!k} P'$.

DEFINITION 15 (WEAK BISIMULATION FOR CLOSED PROCESSES). $\mathcal{R} \subseteq \mathbb{P}_{cl} \times \mathbb{P}_{cl}$ is a weak bisimulation if whenever $P \mathcal{R} Q$,

- (i) if $P \xrightarrow{w;k} P'$ then $\exists Q'$ such that $Q \xrightarrow{\tau!^* w;k\tau!^*} Q'$ and $P' \mathcal{R} Q'$,
- (ii) if $Q \xrightarrow{w;k} Q'$ then $\exists P'$ such that $P \xrightarrow{\tau!^* w;k\tau!^*} P'$ and $P' \mathcal{R} Q'$ \square

The largest weak bisimulation is an equivalence, denoted \approx .

Bisimulation here is formally similar to its CCS counterpart, but the effects are different. As with CBS, $\tau!_k P \not\approx P$ in general! Let $P \stackrel{\text{def}}{=} ?\lambda x. x!_2 o$ and $Q \stackrel{\text{def}}{=} \tau!_3 P$. Then $P \not\approx Q$, since P will always echo its input, but Q may fail to do so: $Q \xrightarrow{5?2} Q$. This cannot be matched by P since it has to receive, and become $5! o \not\approx Q$.

Also, $\tau!_{k'} v!_k P \approx v!_k P$ only if $k' \leq k$. If $k' > k$ the left hand process will tolerate a transmission of priority k' , but the right won't. Earlier definitions of bisimulation for PCBS prescribed that τs preceding the matching action had to be of equal or higher priority. Interestingly, the apparently more liberal definition above captures the same effect. [Pra93b] devises tests to tell unequal processes apart, and suggests that \approx is therefore a meaningful equivalence for CBS. The same argument can be made for PCBS. Because sums are guarded, the following pleasant property holds.

PROPOSITION 16. \approx is a congruence for PCBS. \square

6. VALUE PRIORITY

A very similar calculus results if a fixed priority function $\zeta: \alpha \rightarrow \mathbb{N}$ is associated with the transmitted data type. Translation functions can now alter the priorities of actions. For the calculus to be well behaved, it is sufficient that translation should preserve the priority order strictly (i.e., be monotonic, and maintain inequalities), and that $\zeta(\phi_1 \phi^\dagger v) = \zeta v$. This confirms that the deprioritisation operator of PCBS, which seems anyway adequate, is as powerful as it can be.

7. CCS WITH PRIORITIES

The difficulties in adding priorities to CCS can be traced to the fact that handshake communication makes an autonomous action out of two controlled ones.

[CH88] begins with an a priori semantics which labels transitions with actions and their priorities. The second stage then says that prioritised actions are unconstrained, but unprioritised actions can only take place if they are not preempted by prioritised τ 's—the negative premise. The resulting new axiom is $a. P + \tau'. Q \sim \tau'. Q$ where τ' is a prioritised τ , the cognate of Law 1(c) of Proposition 11. But the result is not entirely satisfactory even at the cost of a two stage operational semantics, for $a. P + b. Q \not\sim b. Q$ if b is of higher priority than a —the basic difficulty mentioned in the introduction. The authors of [CH88] say that such a possibility would be useful; they also point out that then such actions cannot be restricted! This is precisely the scenario of PCBS.

Defining weak bisimulation for this model is non-trivial [CC93]; it is done by abstracting τs from sequences of actions in a priority sensitive way. (Remember that τs preceding a matching action in PCBS have to be of equal or higher priority).

Programming notation	Calculus notation
*a	α
Option *a = None + Some *a	If $w \in \alpha_\tau$ then $w = \tau$ or $w = v \in \alpha$
say P t = Resp (w,k) P' for some t.	$P \xrightarrow{w!k} P'$
say P t = No_Resp for all t.	$\beta w, k, P'. P \xrightarrow{w!k} P'$
NIL	\circ
SAY (e,k) P	$e!_k P$
LISTEN f	$?f$
TALK (e,k) P f	$e!_k P + ?f$
PAR P_1 P_2	$P_1 P_2$
PARS [P_1; ...; P_n]	$P_1 \dots P_n$
TRANS (k,g;h) P	ϕP where $k_\phi = k$, $\phi^\uparrow = g$ and $\phi_\downarrow = h$
SAYS k [e_1; e_2; ...] P	$e_1!_k (e_2!_k (\dots P))$
BOX (g,h) [P_1; ...; P_n]	$\phi (P_1 \dots P_n)$ where $\phi^\uparrow = g$ and $\phi_\downarrow = h$

TABLE 2. Correspondence between programming and calculus notation

[CW91] presents a priority sum $+'$ similar to Occam's PRIALT; $a. P +' b. Q$ can perform a b only if the environment will not do \bar{a} . Now it is not clear whether $((a. P +' \bar{b}. Q) | (b. R +' \bar{a}. S)) \setminus \{a, b\}$ can perform a τ . Actions are therefore separated into input and output and the initial actions of a prisum are required to be inputs. This breaks the symmetry of pure CCS, and complicates the syntax.

The semantics is complicated: the transition relation is parameterised by the output actions the environment can do, as is the definition of bisimulation. The expected negative premise for $P +' Q$ turns up as a side condition that the actions P can accept, computed independently of the transition system, should not be offered by the environment. Lastly, a ready function is needed to adjust the environment parameter upon communication in a parallel composition.

It is almost as though [CW91] declared output actions autonomous, input actions controlled, and dealt semantically with what could (autonomously) happen. Unfortunately for this interpretation, output actions can be restricted. Worse, τ is independent of the environment as are output actions, but is classified as input, since it can be an initial action in a prisum! Since [CW91] deals with process priority rather than action priority, comparison with PCBS cannot be exact. But it does appear that the complexity of [CW91] comes from the handshake model.

8. A PCBS SIMULATOR IN LAZY ML

Table 2 shows the correspondence between CBS notation and the Lazy ML (LML for short) programming notation [AJ93]. Types and constructors are capitalised, while processes and process constructors are entirely in upper case.

The process type, Proc *a, is given in Table 3. Recursion and conditionals are not part of it. These are taken from LML. The constructor TRANS uses an existential type *b that does not occur on the left hand side. The general guarded sum is not needed for programming, only TALK.

To run a process, apply say to it. Basically, say (SAY (3,1) P) = Resp (3,1) P, and say (LISTEN f) = No_Resp. But if $R \stackrel{\text{def}}{=} 3!_0 P | 5!_0 Q$ then

$$R \xrightarrow{3!0} P | 5!_0 Q \quad R \xrightarrow{5!0} 3!_0 P | Q$$

```

rec type Option *a = None + Some *a
and type Proc *a = LISTEN (*a -> Proc *a)
    + TALK (*a # Int) (Proc *a) (*a -> Proc *a)
    + PAR (Proc *a) (Proc *a)
    + TRANS (Int # (*b-> Option *a) # (*a-> Option *b)) (Proc *b)

and type Response *a = No_Resp + Resp (Option *a # Int) (Proc *a)
and say :: Proc *a -> OracleTree -> Response *a
and pipe :: Proc *a -> Proc *a -> List OracleTree -> List *a

and says p (ot. ots) = case say p ot in
    Resp (None, k) p' : says p' ots
    || Resp (Some v, k) p' : v. says p' ots
    || _ : []
end

and hole x = Some x
and wall x = None

```

TABLE 3. The user interface of the simulator and some abbreviations

Clearly, **say** has to be nondeterministic. One way [Bur88] to achieve nondeterminism with functions is to put the nondeterminism in the data. **say** is given an extra boolean argument, an **oracle**, whose value will be determined at run time, but once fixed will not change. The oracle chooses between the parallel components. Thus for some oracles **t** and **t'**, we have

```

say R t = Resp (3,0) (PAR P (SAY (5,0) Q)) and
say R t' = Resp (5,0) (PAR (SAY (3,0) P) Q).

```

say uses a tree of oracles rather than a single oracle, because after choosing the right branch in $P \mid (Q \mid R)$, a further choice has to be made.

say is usually used packaged up as **says**, also in Table 3. An infix “.” is LML notation for “cons”. **says** produces one trace of transmissions from the process, filtering out τ ’s, and using a list of oracletrees to resolve choices along the way. It can return a finite list if the process has an evolution to one with no transmissions.

The most common interface function in this paper is **pipe** **p** **t** which puts **p** and **t** in communication with one another, but returns only whatever **p** says. It is as though the user were in the same room as **p** when it is on the phone to **t**. In applications, **t** can be the input or the insides of a system, and **p** the output or front end. It is implemented as **says** $\phi(\phi_p p \mid \phi_t t)$ with simple translators, and frequently allows a simple data type when **says** would need separate constructors to distinguish **p**’s utterances from **t**’s.

The simulator is ultimately intended to run on a parallel implementation of LML, where the first process to speak is chosen. Then the process $3!o \mid \perp!o$ will always produce 3 before looping. For now, the list of oracletrees is generated either by **randtrees** **r**, where **r** is the seed for a random number generator, or by **firsttrees** where the choice is always the left component. The latter is useful if the system is essentially deterministic. These pseudo-oracles might make the wrong choice with $3!o \mid \perp!o$ and produce no output at all.

9. PROGRAMMING EXAMPLES

Let I be a set of numbers. Then $\prod_{i \in I} i!$ sorts I . The number of priority levels this program needs depends on I . Hardware implementations typically provide only a limited number of priority levels, so the programs that follow use only a bounded number of them, independent of the input.

Two similar disciplines that are followed both here and in [Pra93c] are that the memory needed by any process, and the size of the transmitted values, are both independent of the input data (counting integers as one “word”, and making other such traditional assumptions). This allows the number of messages to be used as a measure of time taken, and the number of processes as a measure of space.

Neither complexity nor correctness are dealt with in these examples; their purpose is to demonstrate the power of the language. Some proofs were sketched in [Pra93b, Pra93c] and it is clear that the usual techniques of process calculus apply. The examples all use value passing, and the formal integration of proofs about processes with proofs about data is still being worked out.

EXAMPLE 17. [Atomic sequences; last element of a list]

Here is a simple indication that PCBS is strictly more powerful than CBS. Pure CBS, where the data type is the singleton set, is essentially useless: there is no obvious way to get information in and out of systems. Unary coding is not possible because silence cannot be detected. But pure PCBS is useful, because silence at level 0 can be detected by successful transmission at level 1. Thus data can be transmitted in atomic sequences at level 0, with level 1 used as an end marker.

```
rec TRAP n = TALK (n,1) NIL (\x. TRAP x)
and last l = pipe (LISTEN (\x. TRAP x)) (SAYS 0 1 NIL) firsttrees
```

`last` returns $[]$ if `l` is empty, and a singleton list with the last element of `l` otherwise. This cannot be done within CBS. However, `says` and `pipe` can detect termination (in this and other simple cases), signalling it by returning a finite list. See [Pra93c] for examples of use of this kind of termination detection. \square

EXAMPLE 18. [TALK is an interrupt operator]

The motivating example from [CH88] and [CW91] is a counter that accepts “Up” and “Down” commands until interrupted. Here is a variation, a clock that counts off intervals until interrupted by any signal at priority 0 or 1.

```
rec CLOCK n = TALK (n,1) (CLOCK (n+1)) (\x. NIL)
```

To be sure to stop the clock, the signal from outside must be sent at priority 0. \square

EXAMPLE 19. [Primes in increasing order: two processes]

```
rec GEN n = SAY (n, 1) (GEN (n+1))
and TRAP = LISTEN (\p. SAY (p, 0) (TRANS (0, hole, f p) TRAP))
and f p n = if n%p = 0 then None else Some (n)
and primes1 = pipe TRAP (GEN 2) firsttrees
```

`TRAP` hears only primes. For each prime, it wears a further translation layer to hide all multiples of this number. Without priorities, `GEN` would have to wait to hear from `TRAP` before announcing the next number, and `TRAP` must make an announcement for every number, so a type `Composite + Prime Int` would be needed. \square

EXAMPLE 20. [Primes in increasing order: new process per prime]

```

rec GEN n = SAY (n, 1) (LISTEN (\x. if x<n then GEN (n+1)
                                else PAR (PRIME n) (GEN (n+1))))
and PRIME n = LISTEN (\x. if x%n = 0
                        then TALK (n,0) (PRIME n) (\x. PRIME n)
                        else LISTEN (\x. PRIME n))
and TRAP = LISTEN (\n. TALK (n,1) TRAP
                     (\x. TRAP))
and primes2 = pipe TRAP (GEN 2) firsttrees

```

TRAP optimistically tries to declare every number to be a prime. On composite numbers, one of the prime divisors preempts it and the other divisors. GEN can tell primes from composites because the former are echoes of what it said. There is no obvious unprioritised solution corresponding to this one. \square

EXAMPLE 21. [Primes up to n in increasing order: $n + 1$ processes]

This program lists primes in the range $[2..n]$, by systematically testing each p for divisibility by all numbers less than itself. Without priorities, the primes will not come out in order. For the same price, a faster program can be had by reversing the priorities and replacing $i*i>$ by $i=$ in the tests.

```

rec GEN i n = SAY (i, 1) (if i=n then NIL else GEN (i+1) n)

and CELL p = LISTEN (\i. if i=p then SAY (p,0) NIL
                      else if p%i = 0 then NIL
                      else CELL p)
and primes3 n r = pipe (PARS (map CELL [2..n])) (GEN 2 n) (randtrees r)

```

\square

EXAMPLE 22. [Primes up to n : possibly only two broadcasts per prime]

```

rec CELL n = TALK (n, 0) NIL (\p. if p ~= n & n%p = 0 then NIL
                                else CELL n)
and primes4 n r = pipe (TRANS (1, hole, hole) (PARS (map CELL [2..n])))
                           (PARS (map CELL [2..n]))
                           (randtrees r)

```

This is a two phase algorithm. In the first phase, all primes and some of the composite numbers on the far side of the pipe announce themselves. Numbers die when they hear a divisor. Survivors on the near side announce themselves in the second phase. The oracles decide which composites speak in the first phase, and in what order the primes speak in the second.

A data type could be used to tell first phase transmissions from second. The first are to be hidden. The code here uses `pipe` instead to get away with processes of type `Int`. The penalty is that there are twice as many processes.

The translator allows `CELL` to be reused, instead of a new process that speaks at priority 1. The deprioritisation has the further advantage that competing processes have priority 0 also in the second phase (inside the translator). The arbitrator has to choose some speaker of highest priority. It helps if this is 0, for then it may have to look no further. If the second phase `CELL`s each spoke at level 2, the arbitrator would have to check them all to make sure no one wishes to speak at 1 or 0. \square

EXAMPLE 23. [Fibonacci numbers]

```

rec ADD n = SAY (n,0) (LISTEN (\v. (ADD (n+v))))
and BUF n = LISTEN (\m. SAY (n,0) (BUF m))
and fib1 s = pipe (SAY (0,0) (ADD 1))
    (LISTEN (\v. BUF v))
    (randtrees s)
and fib2 = pipe (SAY (0,0) (ADD 1))
    (SAYS 1 fib2 NIL)
    firsttrees

```

fib1 is an unprioritised solution. **fib2** is a Kahn network. The output is fed back at priority 1, and so has to synchronise with the adder's pauses. **fib2** is sure to work only with **firsttrees**. Otherwise, it can get into a loop should the oracle choose the output as speaker when it is not ready. \square

EXAMPLE 24. [Broadcast sort]

This is a parallelised insertion sort. The input list is made into a process by **SAYS** and fed into **SORTER**. The input so far is held in a sorted list, maintained by **CELLs** each holding a number n and a “link” l , the next lower number. The next input number splits exactly one cell into two. **BOT** and **TOP** are the end versions of **CELL**. The output phase is initiated by **BOT** succeeding in transmitting the head of the sorted list at low priority. Each cell (l, n) transmits n when it hears l , at high priority if $l = n$. This last is crucial. Otherwise, the input list [1;2;2;3] would be sorted into PARS [BOT 1; CELL 1 2; CELL 2 2; CELL 2 3; TOP 3] and the output could come out as [1;2;3;2] since both (2,2) and (2,3) are triggered by 2.

```

rec type Act = In Int + Out Int
and type React = Split Int + Skip + Say

and comp (In m) l n & ((l<m) & (m<=n)) = Split m
|| comp (Out m) l n & (m=1) = Say
|| comp _ _ _ = Skip

and BOT n = TALK (Out n, 1) NIL (\v. let In m = v in
                                if m<=n then PAR (CELL m n) (BOT m)
                                else BOT n)

and TOP l = LISTEN (\v. case v in
                        In m: if m>l then PAR (CELL l m) (TOP m)
                        else TOP
                        || _ : NIL
                        end)

and CELL l n = LISTEN (\v. case comp v l n in
                            Say : if l=n then SAY (Out n, 0) NIL
                                              else SAY (Out n, 1) NIL
                            || Split m: PAR (CELL l m) (CELL m n)
                            || Skip : CELL l n
                            end)

and SORTER = LISTEN (\v. let In m = v in PAR (TOP m) (BOT m))

and mapin = map (\n. In n)
and mapout = map (\v. let Out m = v in m)
and bsort l r = mapout (pipe SORTER (SAYS 0 (mapin l) NIL) (randtrees r))

```

In CBS, duplicates either have to be dropped, or kept track of by a third parameter to **CELL**. To see how a proof of correctness would go, see [Pra93b]. \square

EXAMPLE 25. [Distributed backtrack: The eight queens problem]

This program is not limited to queens (redefine the function `check`) or to any relation between the size of the board and the number of pieces that can be placed.

```

rec type Act = Place Sq + Revoke + Board (List Square)

and FREE sq = TALK (Place sq, 0) (PLACED sq 1)
    (\x. let Place p = x in
        if check sq p then CHECKED sq 1 else FREE sq)

and PLACED sq 1      = TALK (Revoke,1) (CHECKED sq 1) (\x.PLACED sq 2)
|| PLACED sq rank = BUSY sq rank PLACED

and CHECKED sq 0      = FREE sq
|| CHECKED sq rank = BUSY sq rank CHECKED

and BUSY sq rank state = LISTEN (\x. if x = Revoke then state sq (rank-1)
                                else state sq (rank+1))

and QUEENS size = PARS [FREE (i,j);; i <- [1..size]; j<- [1..size]]

```

Each square is a process, `FREE` to start with. Free squares try to grab a piece; one succeeds. As a result, others may become `CHECKED` and will then await a `Revoke` by the square that checked them. It will eventually happen that no free square remains, perhaps before the maximum number of pieces have been placed. This is detected by the last placed piece, which succeeds in doing a low priority `Revoke`. A revoked square pretends it was checked by the previous piece, thus avoiding looping.

The program puts out sequences of `Places` and `Revokes` till all solutions have been exhausted. It is equivalent to a sequential one with a stack; `PLACED` squares say where they are on this stack. Because of the non-determinism, the first solution can often be found very quickly, with few or no `Revokes`. Finding all solutions takes the same total time no matter how the non-determinism is resolved.

```

and ARCHIVE l max =
  LISTEN (\x. case x in
    Place' p : ARCHIVE (p, 1) max
    || Revoke' : if length l >= max
      then SAY (Board l, 0) (ARCHIVE (tl l) (length l))
      else ARCHIVE (tl l) max
    end)
and queens size r = pipe (ARCHIVE [] 0) (TRANS (1, hole, wall) (QUEENS size))
                           (randtrees r)

```

`ARCHIVE` is one of many possible printer processes. It prints all maximal solutions, and all locally maximal ones prior to the first. It prints at a higher priority to avoid loss of information from `QUEENS`. Note the use of deprioritisation to avoid adjusting the priorities in `QUEENS`, and to keep 0 as the highest priority in it. \square

EXAMPLE 26. [Distributed search: Root of a monotonic function]

Given a monotonic function `f: Float -> Float`, and a `y: Float`, binary search is a sequential way to find the `x` within `range` such that `f x = y` to an accuracy given by `enough`. (If there is no such `x`, the program loops).

The program below parallelises the search by dividing the range into n sections. **CELL** (k, n) computes $f x$, where x is the midpoint of the k th section. When a process converges to $f x$, it reports this value, and **range** is adjusted. This may result in some processes finding that their x is no longer inrange, in which case they abandon their current computation, and start a fresh one.

This description assumes computations can be interrupted. Otherwise, each round of computation has all processes ready to report, and all but the two nearest the root will find that they have completed a useless computation. So processes come up for communication several times during a computation, which is divided into grain sized bits. Before each round, **newst** is computed from **state** (the function **f** is built into **next**). If a **CELL** has nothing to report for that round, it **WAITS**. If it does have something **TOSAY**, it will do so, unless it hears a better value. When no one has anything more to say, they hear a **Tick** from **TRAP**, and continue with their computations.

Without priorities, every process would have to report after each round.

To avoid long lists of parameters, several have been built into the program below. A running version should have **TOSAY** speaking at priority 0, and deprioritise the array of **CELLS**.

```

rec type Act = Tick + Rpt State Bool + Done

and TRAP y = TALK (Tick, 2) (TRAP y) (\b. if enough b y then SAY (Done, 0) NIL
                                         else TRAP y)

and TOSAY id b y range =
    TALK      (b, 1)           (WAIT id Fresh      y (adjust range b))
    (\b'. case b' in
        Done :                  NIL
        || _ : if better b' b then WAIT id Fresh      y (adjust range b')
                           else TOSAY id b      y (adjust range b')
    end)

and WAIT id state y range =
    LISTEN (\b. case b in
              Tick : CELL id state      y range
              || Done : NIL
              || _ : WAIT id state      y (adjust range b)
    end)

and CELL id state y range &                                (inrange range state)=
    let newst = next state grain in
    if converge newst state
    then let b = y_of newst>=y in TOSAY id (newst, b) y range
    else                      WAIT id newst      y range
    || CELL id _ y range =
        let state = fresh range id in CELL id state      y range

and search y range n r =
    says (PARS (TRAP y. (map (\k. CELL (k,n) Fresh      y range) [1..n])))
          (randtrees r)

```

□

Of the examples in this section, the first two might be called “concurrent” programs, and the rest “parallel”. (P)CBS has an implementation on top of a quasi-parallel evaluator [RW93], which shows parallelism profiles. This shows that most of the programs here are so fine grain that communication overheads drown the parallelism, at least in this implementation. The CBS formulation is then valuable for its structure rather than parallelism. By contrast, the distributed search above allows significant parallelism and meaningful experiments in optimisation and load balancing by varying `grain` and `n`.

10. IMPLEMENTATION; INTEGRATION OF EVALUATION AND COMMUNICATION

[Pra93c] presents an extended calculus with both evaluation and communication transitions that is the basis for an implementation by a simulator in a functional language. Evaluation is borrowed from the host language, and accounts for recursion and conditionals. All of this can be extended to PCBS. A study of the implementation, including a proof that the concrete treatment of τ 's is equivalent to the abstract one, is left to a forthcoming report.

11. CONCLUSIONS

Priorities can be added easily to CBS, and significantly extend its power.

The transition system for PCBS yields only those transitions that can actually take place. It differs from that of CBS primarily in the side conditions for the prefix rules. The only other material change is the possibility of deprioritisation.

PCBS is simple because only autonomous actions need be considered when deciding which process should act next. These actions are distinct ones to which it is meaningful to assign priorities. By contrast, it is precisely the fact that there is only one autonomous action in CCS that makes it difficult to add priorities to it. Encoding a negative premise about transmission as a positive one about reception compacts the system further.

Strong and weak bisimulations in PCBS are congruences, and yield observationally meaningful equivalences. Putting deprioritisation aside, only only law distinguishes PCBS from CBS: outranked output guards in a sum cannot be used. This corresponds to a similar law in [CH88]. Interestingly, neither the expansion theorem nor the law of distribution of translation over $|$ need change.

Priorities make a useful addition to the CBS paradigm. They provide atomic sequences (detecting termination, absence of response, etc.), preemption of default actions by exceptions (or interrupts), and a means of doing some actions immediately. Proofs were dropped in favour of more examples, but it should be clear that the usual techniques of process calculus are applicable.

[Pra93b, Pra93c] suggest that CBS could be a practical programming language. This paper records a further step, experiments with a parallelism profiler.

12. RELATED AND FUTURE WORK

[Pra93a] develops a Timed CBS. It shows that the same aspects of broadcast communication are again useful, and that PCBS can be derived from Timed CBS. Most of Timed CBS can be derived from PCBS. [Jef92] suggests that a (discrete) timed calculus can be derived from a prioritised calculus; one relevant aspect here is that PCBS appears to extend to dense priorities with no change.

Adding probabilities to CCS (CBS) is difficult (easy) for the same reasons as adding priorities is.

Future work for CBS includes the development of a proof system, further development of implementations, and comparison with such systems as LINDA [CG89], GAMMA [BM91] and I/O automata [LT87]. These will be extended to PCBS.

Statecharts [Har87] and ESTEREL [BG92] both use broadcast communication, and deal with interrupts and timeouts. However, the models are different enough that comparison is difficult. ESTEREL, for example, allows multiple signals to be broadcast simultaneously, and the receiver chooses which to act on.

Acknowledgements. I thank members of the concurrency and functional programming groups at Chalmers for support and encouragement.

REFERENCES

- [AJ93] Lennart Augustsson and Thomas Johnsson. Lazy ML user's manual. Technical report, Department of Computer Science, Chalmers University of Technology, 1993.
- [BA90] M. Ben-Ari. *Principles of Concurrent and Distributed Programming*. Prentice-Hall, 1990.
- [BBK85] J. C. M. Baeten, J. A. Bergstra, and J. W. Klop. Syntax and defining equations for an interrupt mechanism in process algebra. Technical Report CSR8503, Centre for Mathematics and Computer Science, Amsterdam, 1985.
- [BG92] Gérard Berry and Georges Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19, 1992.
- [BM91] J.-P. Banâtre and D. Le Metayer, editors. *Research Directions in High-Level Parallel Programming Languages*. Springer Verlag LNCS 574, 1991.
- [Bur88] F. W. Burton. Nondeterminism with referential transparency in functional languages. *The Computer Journal*, 31(3):243–247, 1988.
- [CC93] Linda Christoff and Ivan Christoff. Observational equivalence for processes with priorities. Technical report, Dept. of Computer Systems, Uppsala Univ., 1993.
- [CG89] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.
- [CH88] Rance Cleaveland and Matthew Hennessy. Priorities in process algebras. In *Symposium on Logic in Computer Science*. IEEE, 1988.
- [CW91] Juanito Camilleri and Glynn Winskel. CCS with priority choice. In *Symposium on Logic in Computer Science*. IEEE, 1991.
- [Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8, 1987.
- [Jef92] Alan Jeffrey. Translating timed process algebra into prioritised process algebra. In *Nijmegen Symposium on Real-Time and Fault-Tolerant Systems*, 1992.
- [LT87] Nancy Lynch and Mark Tuttle. Hierarchical correctness proofs for distributed algorithms. Technical Report MIT/LCS/TR-387, Laboratory for Computer Science, Massachusetts Institute of Technology, 1987.
- [Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.
- [Pra91] K. V. S. Prasad. A calculus of broadcasting systems. In *TAPSOFT'91 Volume 1: CAAP*, April 1991. Springer Verlag LNCS 493.
- [Pra93a] K. V. S. Prasad. Broadcasting in time. Technical report, Department of Computer Science, Chalmers University of Technology, 1993.
- [Pra93b] K. V. S. Prasad. A calculus of value broadcasts. In *PARLE'93*, June 1993. Springer Verlag LNCS 694.
- [Pra93c] K. V. S. Prasad. Programming with broadcasts. In *CONCUR'93*, August 1993. Springer Verlag LNCS 715.
- [RW93] C. Runciman and D. Wakeling. Profiling parallelism. Internal report, Department of Computer Science, University of York, 1993.

Towards Unifying Partial Evaluation, Deforestation, Supercompilation, and GPC

Morten Heine Sørensen, Robert Glück & Neil D. Jones

Authors' address:** DIKU, Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark, E-mail:
`rambo@diku.dk`, `glueck@diku.dk`, `neil@diku.dk`

Abstract. We study four transformation methodologies which are automatic instances of Burstall and Darlington's fold/unfold framework: *partial evaluation*, *deforestation*, *supercompilation*, and *generalized partial computation (GPC)*. One can classify these and other fold/unfold based transformers by how much information they maintain during transformation.

We introduce the *positive supercompiler*, a version of deforestation including more information propagation, to study such a classification in detail. Via the study of positive supercompilation we are able to show that partial evaluation and deforestation have simple information propagation, positive supercompilation has more information propagation, and supercompilation and GPC have even more information propagation. The amount of information propagation is significant: positive supercompilation, GPC, and supercompilation can specialize a general pattern matcher to a fixed pattern so as to obtain efficient output similar to that of the Knuth-Morris-Pratt algorithm. In the case of partial evaluation and deforestation, the general matcher must be rewritten to achieve this.

1 Introduction

Our aim is to compare certain, automatic instances of the Burstall-Darlington framework. As is well-known, the basic techniques of the framework are: *unfold-ing*, *instantiation*, *definition*, *folding*, and *abstraction*. In addition to applying these mechanisms, the transformers we consider maintain information such as the previously encountered terms, the previously encountered tests, *etc.*

Partial evaluation, discussed at length in [Jon93], replaces calls with some arguments known, *e.g.* f_2v , by specialized calls, *e.g.* f_2v , where f_2 is an optimized version of f taking into account that the first argument is known to be 2. In *offline* partial evaluators, application of Burstall-Darlington transformations is guided by automatically generated *program annotations* that tell where to unfold, instantiate, define, and fold.

Deforestation, due to Wadler [Wad88,Fer88], can remove some intermediate data structures altogether, thus reducing the number of “passes” over data.

** This work was supported by the Austrian Science Foundation (FWF) under grant number 10780 and the Danish Research Council.

Perhaps less well known, it can also partially evaluate. For instance, applying deforestation to $a(A : B : x)(C : y)$, where a is the well-known *append* function, yields $A : B : f\,x\,y$ where f is defined as

$$\begin{aligned} f \sqcup ws &\quad \leftarrow C : ws \\ f(v : vs) ws &\leftarrow v : (f\,vs\,ws) \end{aligned}$$

Termination-safe extensions of deforestation [Chi93,Sor94a] use automatically precomputed annotations to tell where to abstract (=generalize=extract) so that enough folding takes place to ensure termination.

Supercompilation is a powerful technique due to Turchin [Tur86] (continuing Soviet work from the early 1970's) which can achieve effects of both deforestation and partial evaluation. Supercompilation performs *driving*: unfolding and propagation of information, and *generalization*: a form of abstraction which enables folding. The decision when to generalize is taken online. Recent work by Glück and Klimov has expressed the essence of driving in the context of a more traditional tail-recursive language manipulating Lisp-like lists [Glu93].

Generalized partial computation (GPC) due to Futamura and Nogi [Fut88] and later applied to a lazy functional language [Tak91] has similar effects and power as supercompilation, but has not yet been implemented.

The remainder of the paper is organized as follows. In Section 2 we introduce some terminology that allows us to discuss the quality of the output of transformers. In Section 3 we introduce the positive supercompiler, discuss its correctness, and point out the differences in its rules compared to deforestation. Following up on this, we compare in Section 4 the notion of information propagation in deforestation with that in positive supercompilation. We observe that the deforestation algorithm cannot derive efficient Knuth-Morris-Pratt style pattern matchers, while the positive supercompiler can, and we explain why. In Section 5 we extend the comparison to partial evaluation, GPC, and traditional supercompilation. The last section concludes and reviews directions for future research.

2 A test for program transformers

A way to test a method's power is to see whether it can derive certain well-known efficient programs from equivalent naive and inefficient programs. One of the most popular such tests is to generate, from a general pattern matcher and a fixed pattern, an efficient specialized pattern matcher as output by the Knuth-Morris-Pratt algorithm [Knu77]. We shall call this *the KMP test*.

2.1 General, naively specialized, and KMP specialized matchers

Two different *general* pattern matchers are at work in the literature: a tail-recursive one, and one with nested calls. We shall be concerned with the tail-

recursive one. Except where we deny it explicitly, everything carries over to the nested version.³

Definition 1 General, tail-recursive matcher.

$\text{match } p \ s$	$\leftarrow \text{loop } p \ s \ p \ s$
$\text{loop } [] \ ss \ op \ os$	$\leftarrow \text{True}$
$\text{loop } (p : pp) \ [] \ op \ os$	$\leftarrow \text{False}$
$\text{loop } (p : pp) \ (s : ss) \ op \ os$	$\leftarrow p = s \rightarrow \text{loop } pp \ ss \ op \ os \ \square \ \text{next } op \ os$
$\text{next } op \ []$	$\leftarrow \text{False}$
$\text{next } op \ (s : ss)$	$\leftarrow \text{loop } op \ ss \ op \ ss$

Consider the *naively specialized* program $f \ u \leftarrow \text{match } AAB \ u$ which matches the fixed pattern AAB with a string u . Evaluation of $\text{match } AAB \ u$, given u , proceeds by comparing A to the first component of u , A to the second, B to the third. If at some point the comparison failed, the process is restarted with the tail of u .

However, this strategy is not optimal *e.g.* if the string u begins with three A 's. It is inefficient to restart the comparsion with the tail $AA\dots$ of u since it is already known that the first two tests of AAB against $AA\dots$ will succeed. The following *KMP-style specialized* matcher corresponding to the DFA constructed by the Knuth-Morris-Pratt algorithm [Knu77] takes this information into account.

Example 1 KMP-style specialized matcher for AAB .

$\text{match}_{AAB} \ u$	$\leftarrow \text{loop}_{AAB} \ u$
$\text{loop}_{AAB} \ []$	$\leftarrow \text{False}$
$\text{loop}_{AAB} \ (s : ss)$	$\leftarrow A = s \rightarrow \text{loop}_{AB} \ ss \ \square \ \text{loop}_{AAB} \ ss$
$\text{loop}_{AB} \ []$	$\leftarrow \text{False}$
$\text{loop}_{AB} \ (s : ss)$	$\leftarrow A = s \rightarrow \text{loop}_B \ ss \ \square \ \text{loop}_{AAB} \ ss$
$\text{loop}_B \ []$	$\leftarrow \text{False}$
$\text{loop}_B \ (s : ss)$	$\leftarrow B = s \rightarrow T \ \square \ A = s \rightarrow \text{loop}_B \ ss \ \square \ \text{loop}_{AAB} \ ss$

2.2 A comment on measuring complexity

One must be careful when discussing complexity of multi-input programs, especially in the context of program specialization when some inputs are fixed. For an example, let π be the general tail-recursive pattern matchers, and let $|p|, |s|$ denote the length of the pattern p and string s , respectively. Let $t_\pi(p, s)$ be the running time of program π on inputs p, s . Finally, let π_p be the result of specializing π to known pattern p by some transformation algorithm.

When π_p is a specialized KMP style pattern matcher, it is customary to say that the general $O(|p| \cdot |s|)$ time program has been transformed to an $O(|s|)$ time

³ We use the Miranda notation for lists, *e.g.* $[A, A, B]$, as well as the short notation AAB . We shall even continue to do so after having introduced a language with slightly different syntax in Section 3.

program. This is, alas, *always true*, even for trivial transformations such that of Kleene's $s - m - n$ Theorem [Kle52]. The reason is that as soon as $|p|$ is fixed, then $O(|p| \cdot |s|) = O(|s|)$, even though the coefficient in $O(\cdot)$ is proportional to $|p|$.

To be more precise, define the *speedup function* as in [Jon93] as

$$\text{speedup}_p(s) = \frac{t_\pi(p, s)}{t_{\pi_p}(s)}$$

Now for any p there is a constant a and there are infinitely many subject strings s such that $t_\pi(p, s) \geq a \cdot |p| \cdot |s|$. Using a trivial specializer as in the $s - m - n$ Theorem it is easy to see that π_p has essentially the same running time as π , so $\text{speedup}_p(s) \approx 1$.

On the other hand, using non-trivial transformers (see later), the program π_p satisfies $t_{\pi_p}(s) \leq b \cdot |s|$ for any subject string s , where b is independent of p . As a consequence

$$\text{speedup}_p(s) \geq \frac{a \cdot |p|}{b}$$

This is particularly interesting because the speedup is not only significantly large, but becomes larger for longer patterns.

We shall say that the KMP test is passed by a transformer when it holds that there is a constant b such that for all s , $t_{\pi_p}(s) \leq b \cdot |s|$.

In Section 4 we investigate the KMP test for deforestation and positive supercompilation. In Section 5 we review and explain the known results of the KMP test for partial evaluation, supercompilation, and GPC, and relate all these results.

3 Positive supercompilation

We first present the object language. Next we describe some notions that are convenient for the formulation of the positive supercompiler. We then define the positive supercompiler. Finally we describe its relation to deforestation as defined in [Fer88], and consider the correctness of positive supercompilation.

3.1 Language

The language below extends that of [Fer88] by the presence of conditionals (equality tests between arbitrary terms).

Definition 2 Object language.

$t ::= v$	(variable)
$c t_1 \dots t_n$	(constructor)
$f t_1 \dots t_n$	(f-function call)
$g t_0 t_1 \dots t_n$	(g-function call)
$t_1 = t_2 \rightarrow t_3 \square t_4$	(conditional)
$d ::= f v_1 \dots v_n \leftarrow t$	(f-function definition, no patterns)
$g p_1 v_1 \dots v_n \leftarrow t_1$	
\vdots	(g-function definition with patterns)
$g p_m v_1 \dots v_n \leftarrow t_m$	
$p ::= c v_1 \dots v_n$	(patterns with one constructor)

As usual we require that left hand sides of definitions be *linear*, i.e. that no variable occurs more than once.⁴ We also require that all variables in a definition's right side be present in its left side. To ensure uniqueness of reduction, we require that each function in a program have at most one definition and, in the case of a *g*-definition, that no two patterns p_i and p_j contain the same constructor. The semantics for reduction of a variable-free term is call-by-name, as realized in Miranda [Tur90] by “lazy evaluation.”

Apart from the fact that the language is first-order there are two obvious and quite common restrictions: function definitions may have *at most one pattern matching argument*, and only *non-nested* patterns. (Methods exist for translating arbitrary patterns into the restricted form [Aug85]). In some examples we assume for simplicity that both the deforestation algorithm and the positive supercompiler can handle multiple pattern matching arguments.

3.2 How to find the next call to unfold

We shall express the positive supercompiler by rules for rewriting terms. The rewrite rules can be understood intuitively as mimicking the actions of a call-by-name evaluator — but extended to continue the transformation whenever a value is not sufficiently defined at transformation time to know *exactly which* program rule should be applied. If the applicable rule is not unique, then sufficient code will be generated to account for every run-time possibility.

For every term t two possibilities exist during transformation. (i) In the first case there are two subcases. If $t \equiv c t_1 \dots t_n$, then transformation will proceed to the arguments (based on the assumption that the user will demand that the whole term's value be printed out), and if $t \equiv v$ transformation terminates.

(ii) Otherwise call-by-name evaluation forces a unique call to be unfolded or a branch in a unique conditional to be chosen. For instance, in the term

⁴ Instead of adding an equality test, one could allow non-linear patterns. This would, however, encumber the formulation of the positive supercompiler algorithm.

$g(f t_1 \dots t_n) t'_1 \dots t'_m$ we are forced to unfold the call to f in order to be able to decide which clause of g 's definition to choose. As another example, in the term $g(f t_1 = [] \rightarrow t_2 \sqcap t_3) t_4$ we are forced to unfold the call to f to be able to decide between the branches. This, in turn, is forced by the need to decide which clause of g 's definition to apply.

In case (ii) the term will be written: $t \equiv e[r]$ where r identifies the next function call to unfold, or the conditional to choose a branch in, and e is the surrounding part of the term. Traditionally, these are the *redex* and the *evaluation context*. The intention is that r is either a call which is ready to be unfolded (no further evaluation of the arguments is necessary), or a conditional in which a branch can be chosen (the terms in the equality test are completely evaluated).

We now define these notions more precisely.

Definition 3 Evaluation context, redex, observable, value.

$$\begin{array}{ll} e ::= [] & \text{Evaluation contexts} \\ | g e t_1 \dots t_n & \\ | e' = t_2 \rightarrow t_3 \sqcap t_4 & (\text{First reduce left of } =) \\ | b = e' \rightarrow t_3 \sqcap t_4 & (\text{Then reduce right of } =) \\ e' ::= e | c b_1 \dots b_{i-1} e' t_{i+1} \dots t_n & (\text{Left to right under constructor in test}) \end{array}$$

$$\begin{array}{ll} r ::= f t_1 \dots t_n & \text{Redex} \\ | g o t_1 \dots t_n & \\ | b_1 = b_2 \rightarrow t \sqcap t' & \end{array}$$

$$\begin{array}{ll} o ::= c t_1 \dots t_n | v & \text{Observable} \\ b ::= c b_1 \dots b_n | v & \text{Value} \end{array}$$

The expression $e[t]$ denotes the result of replacing the occurrence of $[]$ in e by t . A term with no variables is called *ground*.

It is easy to verify that any term t is either an *observable* o , which is a variable or has a known outermost constructor; or it decomposes uniquely into the form $t \equiv e[r]$ (*the unique decomposition property*). This provides the desired way of finding the next function call to unfold or the conditional in which to select a branch.

3.3 The Positive supercompiler

We can now define the positive supercompiler. The positive supercompiler consists of three elements divided into two phases. In the *transformation* phase, *driving* and *folding* is performed. In the *postprocessing* phase, postunfolding is performed. We first describe driving, then folding and postunfolding.

Driving. The driving part of the positive supercompiler is given in Figure 1. It takes a term and a program (the latter not written explicitly as an argument) and returns a new term and a new program. Following is some notation used in the algorithm.

Fig. 1. Positive supercompiler.**Definition 4.**

- (1) $\mathcal{W}[\![v]\!] \Rightarrow v$
- (2) $\mathcal{W}[\![c t_1 \dots t_n]\!] \Rightarrow c (\mathcal{W}[\![t_1]\!]) \dots (\mathcal{W}[\![t_n]\!])$
- (3) $\mathcal{W}[\![e[f t_1 \dots t_n]]\!] \Rightarrow f^\square u_1 \dots u_k$
where
 $f^\square u_1 \dots u_k \leftarrow \mathcal{W}[\![e[t^f \{v_i^f := t_i\}_{i=1}^n]]\!]$
- (4) $\mathcal{W}[\![e[g(c t_{n+1} \dots t_{n+m}) t_1 \dots t_n]]\!] \Rightarrow f^\square u_1 \dots u_k$
where
 $f^\square u_1 \dots u_k \leftarrow \mathcal{W}[\![e[t^{g,c} \{v_i^{g,c} := t_i\}_{i=1}^{n+m}]]\!]$
- (5) $\mathcal{W}[\![e[g v t_1 \dots t_n]]\!] \Rightarrow g^\square v u_1 \dots u_k$
where
 $g^\square p_1 u_1 \dots u_k \leftarrow \mathcal{W}[\![(e[t^{g,c_1} \{v_i^{g,c_1} := t_i\}_{i=1}^n])\{v := p_1\}]\!]$
 \vdots
 $g^\square p_l u_1 \dots u_k \leftarrow \mathcal{W}[\![(e[t^{g,c_l} \{v_i^{g,c_l} := t_i\}_{i=1}^n])\{v := p_l\}]\!]$
- (6) $\mathcal{W}[\![e[b = b' \rightarrow t \square t']]\!] \Rightarrow \mathcal{W}[\![e[t']]\!]$
if b, b' are ground and $b \equiv b'$
- (7) $\mathcal{W}[\![e[b = b' \rightarrow t \square t']]\!] \Rightarrow \mathcal{W}[\![e[t]]\!]$
if b, b' are ground and $b \not\equiv b'$
- (8) $\mathcal{W}[\![e[b = b' \rightarrow t \square t']]\!] \Rightarrow b = b' \rightarrow \mathcal{W}[\![(e[t])MGU(b, b')]\!] \square \mathcal{W}[\![e[t']]\!]$
if if not both b, b' are ground

Notation. Let p be the implicit program argument.

For f -functions, t^f denotes the right hand side of the definition for function f in p , and $v_1^f \dots v_n^f$ are the formal parameters in f 's definition.

For a g -function definition, $t^{g,c}$ is the right side of g corresponding to the left side whose pattern contains the constructor c . Further, $v_1^{g,c} \dots v_n^{g,c}, v_{n+1}^{g,c} \dots v_{n+m}^{g,c}$ are the formal parameters of g in the clause whose pattern contains the constructor c . Here $v_1^{g,c} \dots v_n^{g,c}$ are those not occurring in the pattern (these are the same for all c) while $v_{n+1}^{g,c} \dots v_{n+m}^{g,c}$ are the variables in the pattern (here m depends on c). Finally, $p_1^g \dots p_l^g$ are the patterns of g .

The expression $t\{v_i := t_i\}_{i=1}^n$ denotes the result of replacing all occurrences of v_i in t by the corresponding value t_i . In $(e[t])\{v_i := t_i\}_{i=1}^n$, the substitution is to be applied to all of $e[t]$, and not just to t .

The notation $MGU(b, b')$ denotes the most general unifier $\{v_i := t_i\}_{i=1}^n$, ($0 \leq n$) of b, b' if it exists, and *fail* otherwise. It is convenient to define t *fail* $\equiv t$.

The symbol \Rightarrow denotes evaluation in the metalanguage, i.e. transformation.

For instance, in clause (3) the result of transforming $e[f t_1 \dots t_n]$ is a call to a new function f^\square . This function is then defined with right hand side the result of transforming $t^f \{v_i^f := t_i\}_{i=1}^n$. The symbol “ \leftarrow ” refers to a definition in the object language (the language of definition 2).

The **where** should be read as a code generation command. A term $e[r]$ is transformed into a call to a new function f^\square ; these new functions are collected somehow in a new program.⁵ The variables $u_1 \dots u_k$ in these calls are simply all the variables of $e[r]$. In clause (5) the variable v is not included in $u_1 \dots u_k$.⁶

Folding and postunfolding. We imagine that the name of the new function and the order of the variables in the list of arguments are uniquely determined by $e[r]$, so if $e[r]$ is encountered later on, the same name and argument list are generated. If this happens the function should not be defined again (a *fold* step is performed). More: we shall assume that a fold step is also performed when a call is encountered which is a renaming of a previously encountered call.

After the transformation phase, all f -functions that are called exactly once in the residual program are unfolded.

3.4 Deforestation versus positive supercompilation

We henceforth call the deforestation algorithm \mathcal{S} and the positive supercompiler \mathcal{W} .

The actions of \mathcal{W} can be cast into the fold/unfold framework as follows. In clause (5) the term is transformed into a call to a new *residual* function. This involves a *define* step: a new function is defined; an *instantiation* step: the new function is defined by patterns; an *unfold* step: the body of the new function is unfolded one step; and a *fold step*: the right hand side of the original function is replaced by a call to the newly defined function.

Clauses (3) and (4) are similar except that there is no need for an instantiation step. We might also say that the instantiation step is trivial, regarding a variable as a trivial pattern and f -functions as defined by patterns. The operation of instantiation followed by unfolding of the different branches is called *driving* by Turchin.

Clauses (6),(7) can be understood as unfold steps similar to clauses (3),(4), and clause (8) can be understood as an instantiation step similar to clause (5).

The *essential difference* between \mathcal{S} and \mathcal{W} is found in clause (5): the pattern p_j^g is substituted for *all* occurrences of the variable v in $e[g v t_1 \dots t_n]$. In \mathcal{S} the pattern is only substituted for the occurrence of v between g and t_1 ; if there are more occurrences of v , then v must be included among the parameters $u_1 \dots u_k$ in both the transformed call and the left hand side of the transformed definition.

⁵ We take the liberty of being imprecise on this point.

⁶ As a matter of technicality, the patterns p_j^g and terms t^{g,c_j} in clause (5) must actually be chosen as renamings of the corresponding patterns and bodies for g , and in clause (8) the unifier must be chosen *idempotent* which is always possible, see [Sor94b].

This very important difference implies that in \mathcal{W} we are propagating more information: the information that v has been instantiated. It will turn out that this accounts for the fact that \mathcal{W} , but not \mathcal{S} , is able to derive KMP style pattern matchers.

It is easy to see that the algorithms have identical effects on linear programs.

Note that the distinction in clause (5) is not made explicit in [Fer88] since \mathcal{S} is restricted to “treeless” programs, all of whose right sides are linear (although transformation of non-linear terms is briefly considered in [Fer88]).

In subsequent examples we shall assume that \mathcal{S} has been extended to handle the conditional by adopting rules (6-8) leaving out the substitution in rule (8).

3.5 Operational semantics, efficiency, and termination

There are three issues of correctness for \mathcal{W} : preservation of operational semantics, nondegradation of efficiency, and termination.

First, the output of \mathcal{W} should be semantically equivalent to the input. A proof of preservation of operational semantics, in a reasonable sense, is given in [Sor94b].

Second, the output of \mathcal{W} should be at least as efficient as the input. Since rewriting to a nonlinear right hand side can cause function call duplication, this will not generally hold unless appropriate precautions are taken. The problem is well-known in partial evaluation [Ses88, Bon90, Jon93] and deforestation [Wad88, Chi93]. Essentially the same principles could be applied to \mathcal{W} , see [Sor94b].

Third, \mathcal{W} should always terminate. The algorithm \mathcal{W} does, in fact, not always terminate. Techniques to ensure termination of \mathcal{W} for all programs are studied in [Sor94b].

4 KMP test of positive supercompiler and deforestation

In this section we observe that \mathcal{S} cannot derive the KMP style pattern matcher. We explain why and show that \mathcal{W} can derive a program almost as efficient as the KMP style pattern matcher. The derived program does contain inefficiency. We explain why and suggest how \mathcal{W} can be extended to produce programs exactly as efficient as the KMP pattern matchers.

4.1 Pattern matching with deforestation

The result of applying deforestation \mathcal{S} to the term $\text{match } AAB \text{ } u$ is as follows, assuming post unfolding:

Example 2 Non-improved specialized matcher.

	<i>loop_{AAB} u u</i>
<i>loop_{AAB} [] os</i>	$\leftarrow \text{False}$
<i>loop_{AAB} (s : ss) os</i>	$\leftarrow A = s \rightarrow \text{loop}_{AB} ss os \sqcap \text{next} os$
<i>loop_{AB} [] os</i>	$\leftarrow \text{False}$
<i>loop_{AB} (s : ss) os</i>	$\leftarrow A = s \rightarrow \text{loop}_B ss os \sqcap \text{next} os$
<i>loop_B [] os</i>	$\leftarrow \text{False}$
<i>loop_B (s : ss) os</i>	$\leftarrow B = s \rightarrow \text{True} \sqcap \text{next} os$
<i>next []</i>	$\leftarrow \text{False}$
<i>next (s : ss)</i>	$\leftarrow \text{loop}_{AAB} ss ss$

This program is only improved in the sense that the p argument has been removed.⁷ But each time a match fails, the head of the string is skipped, and the match starts all over again.

4.2 Pattern matching with the positive supercompiler

We can draw a graph of terms that \mathcal{W} encounters when applied to $\text{match} AABu$, see Figure 2.

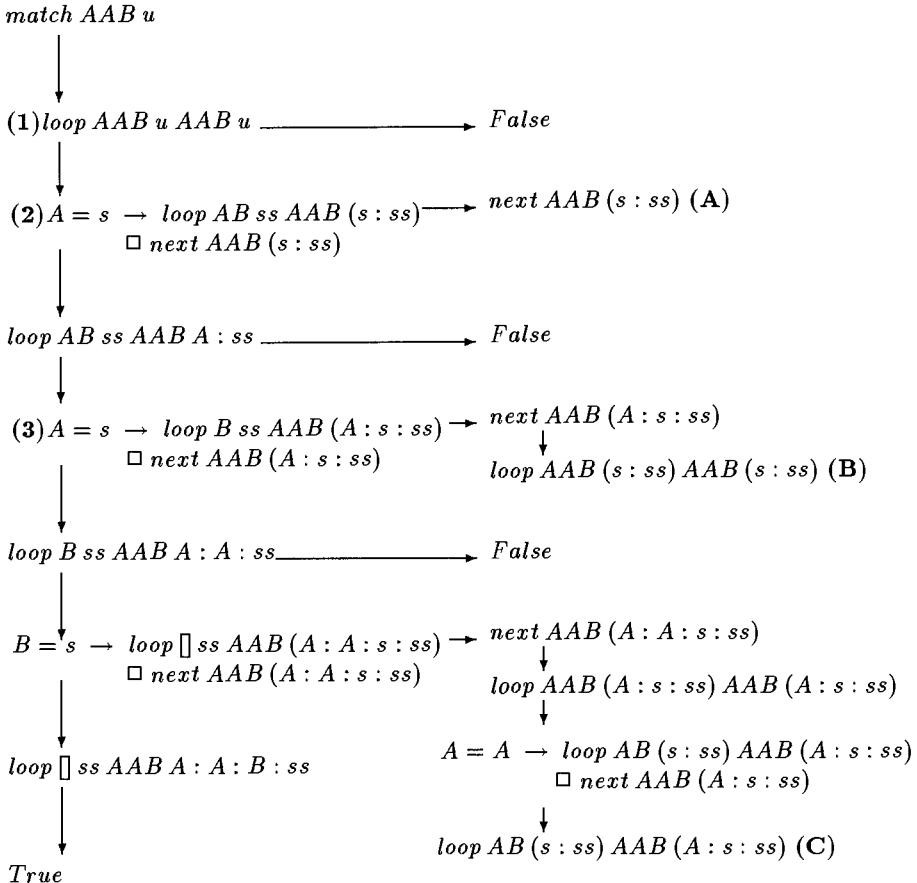
The nodes labelled (A)-(C) in the right column signify that the transformation terminates since the next term has previously been encountered (they signify arcs back to the nodes labelled (1)-(3), respectively, in the left column).

Notice how the instantiation of *all* occurrences of u and ss allows information to be passed to *next* above (1), (2), (3); this effect was not achieved by \mathcal{S} due to the fact that it instantiates only one occurrence of u . These calls to *next* can then be unfolded and some of the subsequent comparisons can be calculated. Specifically, above (C) it is known that we have a string $(A : A : s : ss)$, where s was not B . In moving one step to the right in the string we thus already know that the first comparison between the head of the string and A will succeed; this is in fact what is calculated above (C). The program generated is:

Example 3 Almost KMP style matcher.

	<i>loop_{AAB} u</i>
<i>loop_{AAB} []</i>	$\leftarrow \text{False}$
<i>loop_{AAB} (s : ss)</i>	$\leftarrow A = s \rightarrow \text{loop}_{AB} ss \sqcap \text{next}_{AAB} ss s$
<i>loop_{AB} []</i>	$\leftarrow \text{False}$
<i>loop_{AB} (s : ss)</i>	$\leftarrow A = s \rightarrow \text{loop}_B ss \sqcap \text{next}_{AB} ss s$
<i>loop_B []</i>	$\leftarrow \text{False}$
<i>loop_B (s : ss)</i>	$\leftarrow B = s \rightarrow \text{True} \sqcap A = s \rightarrow \text{loop}_B ss \sqcap \text{next}_{AB} ss s$
<i>next_{AAB} ss s</i>	$\leftarrow \text{loop}_{AAB} ss$
<i>next_{AB} ss s</i>	$\leftarrow A = s \rightarrow \text{loop}_{AB} ss \sqcap \text{next}_{AAB} ss s$

⁷ Incidentally, this shows that deforestation can partially evaluate.

Fig. 2. \mathcal{W} applied to $\text{match } AAB\ u$.

Disregarding the test $A = s$ in next_{AB} which is known to be false when next_{AB} is called from false-branches of the the same test in loop_{AB} and loop_B , this is the desired KMP pattern matcher. The redundant tests do not affect the asymptotic behaviour of the generated specialized matchers: in the terminology of section 2.2, the first author has proved that \mathcal{W} passes the KMP test [Sor94b]. This is a major improvement of \mathcal{W} over \mathcal{S} .

The reason why the unnecessary tests are not cut off in the graph above is that we are only propagating *positive* information. In clause (8) of \mathcal{W} we propagate information that a test was true to the true branch, that certain variables have certain values. We propagate the information by applying the unifier to the term in the true branch.

However, in the false branch we do not propagate the *negative* information

that the test failed. Such information restricts the values which variables can take. Above nodes (A),(B),(C) in the graph we know that s can not be A , A and B , respectively, but this information is ignored by \mathcal{W} , resulting in the redundant tests in the program. Both positive and negative information can arise from an equality test, but we only propagate the positive information.

The reason why we do not propagate negative information in \mathcal{W} is that it cannot be modeled using substitution (what instantiation should one make to express the fact that v is not equal to w ?) We could incorporate negation into \mathcal{W} using other techniques (see the next section) and thereby obtain exactly the KMP pattern matchers by \mathcal{W} .

There is also another kind of information. A call $g v t_1 \dots t_n$ to a function g defined by patterns can be viewed as a test on v . When we instantiate in clause (5) of \mathcal{W} one might say that we test what v is and propagate the resulting information to each of the branches. Here we also represent our positive information by application of a substitution. There is no notion of negative information arising from such a test. Negative information occurs only in the case of (implicit or explicit) “else” or “otherwise” constructs.

5 Partial evaluation, supercompilation, and GPC

In this section we relate information propagation as used in positive supercompilation and deforestation to that in partial evaluation, supercompilation, and GPC.

5.1 Partial Evaluation of Functional Programs

Partial evaluation as in [Jon93] propagates only simple information, *viz.* the values of static variables. So partial evaluation can specialize programs but is weaker than positive supercompilation, supercompilation or GPC, since it propagates no information obtained from predicates or pattern matching. This explains the result found in [Con89], that partial evaluation does not pass the KMP test on the tail-recursive matcher.

Binding-time improvements. The traditional way to improve the result of partial evaluation is to modify the source programs. These modifications, called *binding-time improvements*, are semantics-preserving transformations of the source program which enable a partial evaluator to propagate more information and to achieve deeper specialization.

Consel and Danvy showed that partial evaluators can be used to derive specialized KMP matchers by an “insightful rewriting” of the tail-recursive matcher [Con89] to improve its binding time separation. In short, when the rewritten matcher encounters a mismatch after k successful comparisons, it starts all over comparing the pattern with the $k - 1$ long tail of the pattern, and only after $k - 1$ successful comparisons will it return to the string. In other words, information propagation has been added to the matcher. The same rewriting suffices for deforestation to produce KMP style matchers.

Interpretive approach. It was shown by Glück and Jørgensen that partial evaluators can pass the KMP test by specializing an information propagating interpreter with respect to the tail-recursive matcher and a fixed pattern [Glu94].

5.2 Supercompilation

As mentioned, the mechanism ensuring the propagation of information in supercompilation is *driving*. Here we shall be concerned with driving as described in [Glu93] for a language with lists as data structures.

Let us, for a moment, think of \mathcal{W} as the generalization of a *rewrite* interpreter: when it unfolds a function call it replaces the call by the body of the called function and substitutes the actual arguments into the term being interpreted. Alternatively, one can think of an *environment based* interpreter which creates bindings of the formal parameters to the actual arguments. Correspondingly, one could imagine an environment based version of \mathcal{W} . This is basically what the supercompiler in [Glu93] is.

Thus, the driving mechanisms in the positive supercompiler and in the supercompiler of [Glu93] are identical with respect to the propagation of positive information (assertions) about unspecified entities, except that the former uses substitution and the latter environments. The technique using environments has the advantage that negative information (restrictions) can be represented as bindings which do not hold, and this is done in [Glu93]. A technique using substitutions does not seem possible.

There does not seem to be any significant difference between using environments or substitution for positive information.⁸ If one applies the supercompiler in [Glu93] using only positive information propagation to the tail-recursive matcher, one gets the same program that \mathcal{W} produces; applying the full driving mechanism of [Glu93] with both positive and negative information propagation yields the desired optimal program as shown in [Glu93].

In [Glu90] Glück and Turchin showed that Turchin's supercompiler could pass the KMP test with the nested general matcher.

5.3 GPC

GPC extends partial evaluation as follows. Whenever a conditional (or something equivalent) testing whether predicate P holds is encountered during the transformation, P is propagated to the true branch and the predicate $\neg P$ is propagated to the false branch. Also, whenever a test is encountered, a theorem prover sitting on top of the transformer tests whether more than one branch is possible. If only one is possible, only that branch is taken. GPC is a powerful transformation method because it assumes the (unlimited) power of a theorem

⁸ In self-application of partial evaluation one does binding-time analysis of the partial evaluator; such an analysis gives better results for the environment-based version because it gives better separation of static and dynamic data.

prover. It was shown in [Fut88] that this information suffices to pass the KMP test on the tail-recursive matcher.

Supercompilation and GPC are related, but differ in the propagation of information. While the latter propagates arbitrary predicates requiring a theorem prover, supercompilation propagates structural predicates (assertions and restrictions about atoms and constructors).

Takano concretized GPC for a particular functional language, *viz.* the same as the one studied in the original deforestation paper [Wad88].

There is one rule which is of particular interest for our purposes.

$$\begin{aligned} G[\![\text{case } v \text{ of } p_1 : t_1 \mid \dots \mid p_n : t_n]\!] E &\Rightarrow \\ &\quad \text{case } v \text{ of } p_1 : G[\![t_1]\!] E_1 \mid \dots \mid p_n : G[\![t_n]\!] E_n \\ &\text{where} \\ &E_i = E \cup \{v \leftrightarrow p_i\} \end{aligned}$$

The E 's are sets of equalities (sets of predicates) which are used in the manner described in more general terms in [Fut88]; concretely, they represent positive information arising from pattern matching. The algorithm actually uses a mixture of substitution based and environment representation of information. There is no need for negative information because the language of [Tak91] has no else-construct (just like g -functions in our language have no otherwise clause).

A related substitution based version is:

$$\begin{aligned} G[\![\text{case } v \text{ of } p_1 : t_1 \mid \dots \mid p_n : t_n]\!] E &\Rightarrow \\ &\quad \text{case } v \text{ of } p_1 : G[\![t_1 \{v := p_1\}]\!] E \mid \dots \mid p_n : G[\![t_n \{v := p_n\}]\!] E \end{aligned}$$

The corresponding rule in deforestation is:

$$\mathcal{S}[\![\text{case } v \text{ of } p_1 : t_1 \mid \dots \mid p_n : t_n]\!] \Rightarrow \text{case } v \text{ of } p_1 : \mathcal{S}[\![t_1]\!] \mid \dots \mid p_n : \mathcal{S}[\![t_n]\!]$$

Modulo syntax, the step from the latter rule to the former rule is exactly the same as the step from \mathcal{S} to \mathcal{W} . It is exactly this step which allows the derivation of KMP style pattern matchers, as mentioned briefly in the context of the language with case constructs in [Con93].

6 Conclusion and future work

We compared the transformation methodologies deforestation, partial evaluation, supercompilation, GPC, and positive supercompilation, the latter being new. We showed which notions of information propagation they share, what their differences are, and that the amount of information propagated is significant for the transformations achieved by each methodology.

We demonstrated how the positive supercompiler, using only positive information propagation, can derive an algorithm comparable in efficiency to the matcher generated by the Knuth-Morris-Pratt algorithm starting from a general string matcher and a fixed pattern. Deforestation and partial evaluation cannot achieve this.

Our results are strong evidence that one should not restrict the application of techniques developed in one field to a particular methodology. On the contrary, their integration is on the agenda. However, a direct comparison is often blurred because of different notations and perspectives. Future work may bring the different methodologies even closer, as outlined below.

Until now we have grouped the transformers according to the amount of information propagation. Another classification criterion is the *handling of nested calls*. Deforestation, positive supercompilation, and Turchin's supercompiler all simulate call-by-name evaluation, whereas partial evaluators simulate call-by-value. It would seem that the strength of transformers depend on the transformers "evaluation strategy." For instance, it is well-known that plain partial evaluation does not eliminate intermediate data structures, whereas all the above call-by-name transformers do. On the other hand all, the above call-by-name transformers, including deforestation, can perform partial evaluation. This idea also seems worthy of an investigation. Some steps have been taken in [Sor94b], but further clarification is needed. Related research includes the idea of deforestation by CPS-translation and call-by-value partial evaluation.

The second author has on several occasions noted the correspondence between supercompilation and *interpretation* of logic programs. The correspondence has been stated quite precisely in terms of SLD-trees and so-called process trees in [Sor94b]. Possible payoffs from such an idea include the application of a variety of techniques from one community in the other; the idea certainly seems worthy of study.

There are also connections between supercompilation and *transformation*, in particular partial evaluation, of logic programs. Unlike the situation in the functional case, partial evaluators for Prolog can derive KMP matchers from general Prolog matchers similar to our tail-recursive matcher [Smi91]. This is because partial evaluators for Prolog propagate information (by unification) in a way similar to that in supercompilation. Possible gains from a detailed correspondence may, again, be significant.

References

- [Aug85] L. Augustsson. Compiling Lazy Pattern-Matching. In *Conference on Functional Programming and Computer Architecture*. (Ed.) J.-P. Jouannaud pp368-381. LNCS 201, Springer-Verlag 1985.
- [Bon90] A. Bondorf. *Self-Applicable Partial Evaluation*. Ph.D. thesis, DIKU-Rapport 90/17, Department of Computer Science, University of Copenhagen, 1990.
- [Bur77] R. M. Burstall & J. Darlington. A Transformation System for Developing Recursive Programs. In *Journal of the ACM*. Vol.24, No.1, pp.44-67, 1977.
- [Chi93] W.-N. Chin. Safe Fusion of Functional Expressions II: Further Improvements. In *Journal of Functional Programming*. 1994, to appear.
- [Con89] C. Consel & O. Danvy. Partial Evaluation of Pattern Matching in Strings. In *Information Processing Letters*. Vol.30, No.2, pp.79-86, 1989.
- [Con93] C. Consel & O. Danvy. Tutorial Notes on Partial Evaluation. In *20th ACM Symposium on Principles of Programming Languages*. Charleston, South Carolina, pp.493-501, ACM Press 1993.

- [Fer88] A. B. Ferguson & P. Wadler. When will Deforestation Stop? *1988 Glasgow Workshop on Functional Programming*. pp.39-56, 1988.
- [Fut88] Y. Futamura & K. Nogi. Generalized Partial Computation. In *Partial Evaluation and Mixed Computation*. Eds. A. P. Ershov, D. Bjørner & N. D. Jones, pp.133-151, North-Holland 1988.
- [Glu90] R. Glück & V. F. Turchin. Application of Metasystem Transition to Function Inversion and Transformation. In *Proceedings of the ISSAC '90*. pp.286-287, ACM Press 1990.
- [Glu93] R. Glück & And. Klimov.. Occam's Razor in Metacomputation: the Notion of a Perfect Process Tree. In *Static Analysis, Proceedings. LNCS 724*. pp.112-123, Springer-Verlag 1993.
- [Glu94] R. Glück & J. Jørgensen. Generating Optimizing Specializers. In *IEEE International Conference on Computer Languages*. IEEE Computer Science Press, 1994, to appear.
- [Jon93] N. D. Jones, C. Gomard & P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International. 1993
- [Kle52] S. Kleene, Introduction to Metamathematics. Van Nostrand, 1952.
- [Knu77] D. E. Knuth, J. H. Morris, V. R. Pratt. Fast Pattern Matching in Strings. In *SIAM Journal on Computing*. Vol.6, No.2, pp.323-350, 1977.
- [Ses88] P. Sestoft. Automatic Call Unfolding in a Partial Evaluator. In *Partial Evaluation and Mixed Computation*. Eds. A.P Ershov, D. Bjørner, N.D. Jones, pp.485-506, North-Holland 1988.
- [Smi91] D. A. Smith. Partial Evaluation of Pattern Matching in Constraint Logic Programming Languages. In *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. Ed. N. D. Jones & P. Hudak, pp.62-71, ACM Press 1991.
- [Sor93] M. H. Sørensen. A New Means of Ensuring Termination of Deforestation with an Application to Logic Programming. In *Workshop of the Global Compilation Workshop in conjunction with the International Logic Programming Symposium*. Vancouver, Canada, October, 1993.
- [Sor94a] M. H. Sørensen. A Grammar-based Data-flow Analysis to Stop Deforestation. In *Colloquium on Algebra in Trees and Programming*. Edinburgh, Scotland, April 1994, to appear.
- [Sor94b] M. H. Sørensen. *Turchin's Supercompiler Revisited. An Operational Theory of Positive Information Propagation*. Master's Thesis, Department of Computer Science, University of Copenhagen, 1994.
- [Tak91] A. Takano. Generalized Partial Computation for a Lazy Functional Language. In *ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation*. Ed. N. D. Jones & P. Hudak, pp.1-11, ACM Press 1991.
- [Tur86] V. F. Turchin. The Concept of a Supercompiler. In *ACM TOPLAS*. Vol.8, No.3, pp.292-325, 1986.
- [Tur90] D. Turner. An Overview of Miranda. In *Research Topics in Functional Programming*. Ed. D. Turner, Addison-Wesley, 1990.
- [Wad88] P. L. Wadler. Deforestation: Transforming Programs to Eliminate Trees. In *European Symposium on Programming. Proceedings. LNCS 300*. pp.344-358, Springer-Verlag 1988.

Algebraic Proofs of Properties of Objects

David Walker

Department of Computer Science
University of Warwick
Coventry CV4 7AL, U.K.

Abstract

A semantics by translation to a process calculus is used as the basis for an investigation of transformations to programs expressed in a parallel object-oriented language. Two concrete examples are studied. In one it is shown that transformations introducing concurrency into the design of a priority queue class do not alter the observable behaviour. In the other, a more delicate relationship between two symbol table classes is established.

1 Introduction

In [5] a development method for parallel programs is proposed. Central to it are the application of program transformations to control the introduction of concurrency into designs and the use of concepts from object-oriented programming to constrain interference. Characteristic of the object-oriented style of programming is the description of a computational system as a collection of *objects* each of which is a self-contained entity possessing data and procedures, or *methods*. A parallel object-oriented program typically describes a highly mobile concurrent system in which new objects are created as computation proceeds and the linkage between components changes as references to objects are passed in communications. Providing adequate, tractable models for such systems is challenging. But the challenge is one which must be addressed if program transformations are to be subject to rigorous justification.

Among the most well-developed work on semantics of parallel object-oriented languages is that on the POOL family [3]. For example in [1] an operational semantics for a member of the family is given while [2] offers a denotational semantics based on metric spaces. An alternative technique was introduced in [13] where semantics for two small parallel object-oriented languages were given by translation to the π -calculus [9], a process calculus in which one can naturally express systems which have changing structure. This work was extended in [14] where a semantics for POOL was given by translation to the polyadic π -calculus [8] and a close correspondence between it and a two-level operational semantics, a reformulation of the semantics of [1], was established. The examples in the present paper are expressed in the language $\pi\alpha\beta\lambda$ of [5]. Semantics for it have been given by translation to the polyadic π -calculus [6] and to the higher-order π -calculus [15] introduced in [12].

The basic entities of the π -calculus and its descendants are *names*. In the (polyadic) π -calculus processes communicate by using names to pass (tuples of) names to one another. In the higher-order π -calculus processes may also pass to one another process abstractions of arbitrarily high order. In the semantics by translation each entity – class, object, method, variable, statement, expression – is encoded as a process (or higher-order abstraction), with the representation of a composite phrase being defined directly in terms of those of its

constituents. Execution of a program is represented by reduction of the process encoding it. In particular, the passing of references between objects is captured naturally as the passing of names between the processes representing them with the treatment of *private* names afforded by the restriction operator of the π -calculus playing a central rôle. An important contribution to the perspicuity of the translations is made by the imposition of a sort discipline [8] which constrains the use of names; and in the semantics using the higher-order π -calculus, each program constructor is represented as a higher-order process abstraction. These refinements help to retain some of the high-level structure of programs.

One advantage of giving programming language semantics by translation to a process calculus whose semantic basis is established is that we are thereby freed from the task of constructing an adequate semantic domain and establishing that it supports constructions necessary to give a faithful interpretation of the language. In the case of parallel object-oriented languages, accomplishing this task can require something of a *tour de force*; see e.g. [2]. But the use of this method of semantic definition offers a second potential benefit and it is this that the present paper begins to explore. A topic of central concern in process calculus has been to give abstract descriptions of the behaviour of processes and to develop techniques for reasoning about this behaviour. By encoding language phrases, and in particular programs, as processes, we can use process calculus apparatus to derive abstract descriptions of the behaviour they express and to reason about it. In this paper we investigate this possibility by studying two concrete examples from [5]. Principal aims of this study are to illustrate the utility of process calculi in a rigorous treatment of such problems, and to contribute to the clarification of the difficulties involved in finding and proving sound general transformation rules for parallel object-oriented programs of the kind proposed in [5].

The process calculus used in this paper is an extension of the polyadic π -calculus with value expressions and conditional agents. We do not carry through in detail here the amalgamation of the well-established theories of calculi in which processes exchange simple data values and the π -calculus. Rather we state the results which we assume of the process calculus; they are all well-known in other settings. As mentioned earlier, data can be encoded as π -calculus processes or, more cleanly, as abstractions in the higher-order π -calculus. The reason for working in the less parsimonious setting is that our aim is to give proofs which are both simple and natural, and coding data such as integers and booleans as π -calculus processes is not helpful in achieving this.

It is not possible in the space available to give full accounts of the programming language, the process calculus, the translational semantics, the examples and the analysis of them. In the following section we introduce the examples. Section 3 contains a condensed account of the calculus and an illustration of the semantics. In Section 4 we describe the analysis of the first example and in Section 5 very briefly sketch that of the second. A fuller account of this work is given in [16].

Acknowledgement. I am grateful to Matthew Hennessy, Cliff Jones and Davide Sangiorgi for comments on this work.

2 The Examples

The first example concerns two integer priority queue classes. The first is sequential while the second can be viewed as being obtained from it by transformations which are intended to introduce some concurrency. The first class definition is as follows:

```
class Q
var V:NAT, P:Q
```

```

method Add(X:NAT)
  if V=nil then (V:=X ; P:=new(Q))
  else if V<X then P!Add(X)
    else (P!Add(V) ; V:=X) ;
  return
method Rem():NAT
  var T:NAT
  T:=V ;
  if not(V=nil) then (V:=P!Rem() ; if V=nil then P:=nil) ;
  return T .

```

A class definition provides a template for its *instances*, the objects of that class. Each instance of Q represents a cell which stores a nonnegative integer in the variable V and a pointer to another cell in the variable P. A priority queue is composed of a chain of cells in which integers are stored in ascending order. The value of the expression new(Q) is a reference to a new object of the class. On creation of a cell the values of V and P are undefined (nil). New cells are created and appended to the chain as integers are added via the method Add which takes an integer parameter and returns no result. The smallest integer held in the queue (or nil if the queue is empty) is returned and removed by the Rem method. Evaluation of the expression P!Rem() involves the invocation in the object to which the value of P is a reference of the method Rem. Execution of the invoking object is suspended until an integer is returned to it, this being the value of the expression; a cell returns its value in the Rem method by executing the statement return T. Similarly, the statement P!Add(X) represents an invocation of the method Add in the object to which the value of P is a reference with the value of X as parameter. Again the activity of the invoking object is suspended until it is released from the rendezvous; in this case no value is returned and the appropriate releasing statement is return. An invariant of a queue of cells is that only the first cell is accessible to other objects. When an Add method is invoked in the first cell its activity is suspended until the integer is inserted into the queue at the appropriate point and a return signal ripples back along the queue to the first cell which then releases the caller from the rendezvous. A similar discipline constrains the Rem method. In $\pi\alpha\beta\lambda$, at most one method may be active in an object at any time: a cell may not accept another method invocation until it has completed execution of the active method body.

The second class definition is as follows:

```

class Q'
var V:NAT, P:Q'
method Add(X:NAT)
  return ;
  if V=nil then (V:=X ; P:=new(Q'))
  else if V<X then P!Add(X)
    else (P!Add(V) ; V:=X)
method Rem():NAT
  return V ;
  if not(V=nil) then (V:=P!Rem() ; if V=nil then P:=nil)

```

Q' differs from Q in that when the Add method is invoked in a cell it executes its return statement immediately, thus freeing the calling object from the rendezvous; the two may thus proceed in parallel. The Rem method acts in a similar way; in this case the local variable T is no longer required.

A $\pi\alpha\beta\lambda$ program consists of a sequence of class definitions together with an indication of which class furnishes the *root object* which alone exists at the beginning of a computation.

Suppose *prog* is a program in which the class *Q* is defined, and suppose *prog'* is obtained from it by replacing the definition of *Q* by that of *Q'* and each mention of *Q* by mention of *Q'*. The functions of this transformation are to increase the scope for concurrency in execution of the program, and to do so without altering its observable (input and output) behaviour. That the first intention is fulfilled will not be argued for here, though evidence that it is is discernible in the argument which we give to show that the observable behaviour of the program is not altered. The translational semantics associates with the programs *prog* and *prog'* agents *P* and *P'*. We may express the property of interest as the assertion that $P \simeq P'$ where \simeq is an appropriate one of the many notions of behavioural equivalence which have been studied in process calculus. We show in Section 4 that $P \simeq P'$ holds when for \simeq we take *weak bisimulation equivalence* [7, 9].

The second example concerns two symbol table classes in which references to objects of some class *A* are associated with nonnegative integer keys. Again the first class is sequential while the second can be viewed as being obtained from it by transformations which are intended to introduce some concurrency. The first class definition is as follows:

```
class T
var K:NAT, V:A, L:T, R:T
method Insert(X:NAT, W:A)
    if K=nil then (K:=X ; V:=W ; L:=new(T) ; R:=new(T))
    else if X=K then V:=W
        else if X<K then L!Insert(X,W)
            else R!Insert(X,W) ;
    return
method Search(X:NAT):A
    if K=nil then return nil
    else if X=K then return V
        else if X<K then L!Search(X)
            else R!Search(X)
```

A table is structured as a binary search tree in which each key occurs at most once. Each instance of *T* represents a node of the tree. Only the root node is accessible to other objects. A node has a key *K*, a value *V* which is a pointer to an object of the class *A*, and pointers *L* and *R* to the left and right subtrees. On creation all have undefined values; this is true also of the variables of leaf nodes. The methods of *T* are *Insert* for inserting a key-value pair and *Search* which returns the reference associated with the key supplied as parameter (if it occurs in the table). Analogously to the behaviour of the sequential priority queue *Q*, when a method is invoked in the root node of a tree its activity is suspended until a return ripples back through the tree to it when it releases the caller from the rendezvous.

The second class definition is as follows:

```
class T'
var K:NAT, V:A, L: T' , R: T'
method Insert(X:NAT, W:A)
    return ;
    if K=nil then (K:=X ; V:=W ; L:=new(T') ; R:=new(T'))
    else if X=K then V:=W
        else if X<K then L!Insert(X,W)
            else R!Insert(X,W)
method Search(X:NAT):A
    if K=nil then return nil
    else if X=K then return V
```

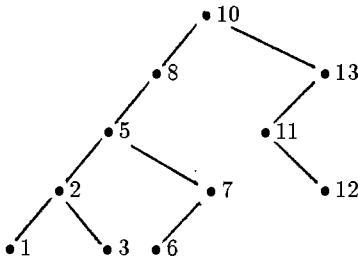
```

else if X<K then commit L!Search(X)
else commit R!Search(X)

```

Two transformations are applied to obtain T' from T . The first involves moving the `return` statement in the method `Insert` to the beginning of the body. The second transformation involves the replacement in the `Search` method of the invocations of `Search` methods in the left and right subtrees by `commit` statements (in [5] ‘yield’ is used instead of ‘`commit`’). When an object α executes a `commit` statement by invoking a method in an object β , it is implicit (i) that β should return its result not to α but to the object γ to which α should return a result, and (ii) that α is freed from the task of returning a result to γ . In particular, execution of α may continue in parallel with that of β . In T' , if the `Search` method is invoked in a node with a key smaller (resp. larger) than that stored there, the node will commit that search to its left (resp. right) child. We may think of the node as passing to the child a return address to which the result of the search should be sent. This address will have been received by the node either directly from the initiator of the search or from its parent in the tree.

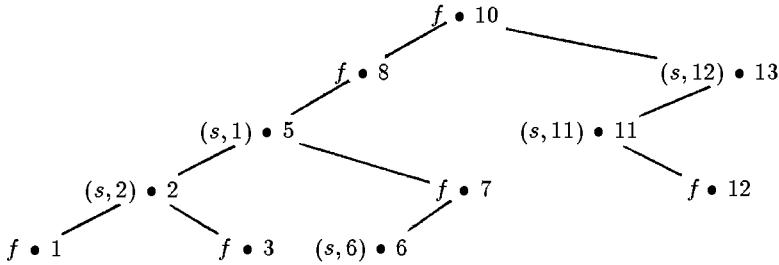
The effect of moving the `return` statement to the beginning of the `Insert` method is similar to that resulting from the analogous transformation to the `Add` method of the class `Q`. More interesting are the transformations which introduce the `commit` statements. A symbol table constructed from nodes of class T' may support a number of concurrent searches. The order in which the results of these searches may be returned is intimately related to the tree structure of the table. For example consider the following tree structure in which no method is active in any of the nodes (the nature of the references associated with the keys is irrelevant to the discussion):



The root node may accept the sequence of invocations

```
Search(7) ; Search(6) ; Search(2) ; Search(11) ; Search(12) ; Search(1)
```

without the result of any of them being returned. Thereafter the results of the searches with keys 7 and 11 may be returned but the results of the searches with the other four keys may not as the progress of each of them is blocked by uncompleted searches in relevant parts of the tree structure. An important point here is that since only one method may be active in an object, one search cannot ‘overtake’ another. If the result of the search with key 7 is returned and each of the searches makes as much progress as is then possible, the state of the table may be pictured thus:



Here we annotate each node with a *status* recording if the node is active and if so how it is engaged. The status is f ('free') if the node is inactive and (s, k) if it is executing a search with parameter k . In this state the results of the searches with keys 2, 6 and 11 may be returned, but the search with key 1 is blocked until that with key 2 returns and the search with key 12 is blocked by the unfinished search with key 11.

It is thus clear that the correspondence between the classes T and T' is not a simple one. Account must also be taken of the progress of invocations of *Insert*. Further, in the case of infinite computations issues related to fairness arise. The analysis we give involves finding an abstract description of the behaviour of the process representing a newly-created table, and showing how the finite parts of this behaviour are related to those of a process describing the behaviour of a newly-created table of class T . Briefly, we show that if μ is a finite sequence of method invocations then the possible computations of the T' -table generated by μ are the parallelizations allowed by the tree structure of the computation of the T -table generated by μ .

3 The Language, the Calculus and the Semantics

The language $\pi\omega\beta\lambda$ in which the examples are expressed is described in [5, 6]. Semantics by translation for it appear in [15], where the higher-order π -calculus is used, and in [6], which employs the polyadic π -calculus. An important development here is the use of an enrichment of the π -calculus with value expressions of simple types and conditional agents. This section contains a condensed description of the calculus and an illustration of the translation of $\pi\omega\beta\lambda$ to it using fragments of the priority queue class Q .

3.1 The Calculus

We assume a set S of *subject sorts* among which are two distinguished sorts N and B for the simple types of natural numbers and booleans. The set O of *object sorts* is S^* , the set of finite tuples of subject sorts. We write (s_1, \dots, s_n) for the object sort of length n consisting of $s_1, \dots, s_n \in S$. A *sorting* is a function $\Sigma : S_0 \rightarrow O$ where $S_0 = S - \{N, B\}$. The basic entities of the calculus are link names and simple value expressions. The names may be thought of as names of channels via which processes may interact. Each name is assigned a subject sort and the sorting determines that only entities with the object sort $\Sigma(s)$ may be communicated using a name of sort $s \in S_0$; these entities are tuples of names and values of the simple types (sorts). The subject sorts N and B have no associated object sorts: entities of these sorts are not names of links. This kind of sorting discipline was introduced in [8]. It was refined in [11] with the adoption of structural matching, rather than name matching, of sorts, and the introduction of input/output tags allowing a natural form of subsorting. Here we use structural matching of sorts but do not require input/output tags.

We assume an infinite set of names x, y, \dots of each subject sort and write $x : s$ to indicate that x is of sort s . We assume that \mathbf{N} contains constant names $0, 1, \dots, \perp$ and \mathbf{B} constant names $\text{tt}, \text{ff}, \perp$, and that both have in addition an infinite number of variable names. A suitably expressive language of value expressions is assumed. We assume that each closed expression (i.e. expression containing no variable names) of sort \mathbf{N} or \mathbf{B} has a value which is a closed name of that sort (with \perp representing an undefined value). We assume also an infinite set of agent constants K of each object sort θ , written $K : \theta$. The set of *agent expressions* P, Q is given by

$$P ::= \Sigma_{j \in J} \pi_j.P_j \mid P \mid Q \mid (\nu x)P \mid K(\tilde{w}) \mid \text{if}(b : P, Q)$$

where: in $K(\tilde{w})$, \tilde{w} ranges over tuples of link names and value expressions; in the conditional agent expression $\text{if}(b : P, Q)$ (“if b then P else Q ”), b ranges over value expressions of sort \mathbf{B} ; a *prefix* π is an output of the form $\bar{x}(\tilde{w})$ or an input of the form $x(\tilde{y})$; in a restriction (νx) , x has a sort in \mathcal{S}_0 ; the indexing set J in the guarded summation is finite; and each constant K has a defining equation $K \stackrel{\text{def}}{=} F$ where F is an *abstraction* of the form $(\tilde{y})P$ where \tilde{y} is a tuple of distinct variable names containing all those occurring free in P (see below). We write $\mathbf{0}$ for an empty summation and $\pi \cdot \mathbf{0}$ for $\pi \cdot \mathbf{0}$. In $x(\tilde{y}).P$ and $(\tilde{y})P$ the occurrences of \tilde{y} are binding with scope P ; the restriction operator (νx) is also a binding operator. We assume the standard notions of free names, substitution (of link names for link names and value expressions for variable names of sort \mathbf{N} or \mathbf{B}), alpha-conversion etc. and identify agent expressions which differ only by change of bound names. We write $\text{fn}(P)$ for the set of free names of P . We consider only *sort-respecting* substitutions, i.e. those σ such that x and $\sigma(x)$ have the same sort for every (variable) name x . We say a substitution σ is *closed* if for each variable name x of sort \mathbf{N} or \mathbf{B} , $\sigma(x)$ is a constant name. An agent expression containing no free variable names of sort \mathbf{N} or \mathbf{B} is called a *process*.

We consider only agent expressions which *respect* the sorting Σ . The essence of respecting Σ , which is defined formally by a family of rules [8], is that each well-sorted agent expression is assigned an object sort, with $P : ()$ for each process P , so that definitions, applications and prefixes are consistent with Σ . Thus extending ‘ $:$ ’ componentwise to tuples we have e.g. that if $K \stackrel{\text{def}}{=} (\tilde{y})P$ and $K : \theta$ then $\tilde{y} : \theta$ and in $K(\tilde{w})$ we must have $\tilde{w} : \theta$; and in $\pi = \bar{x}(\tilde{w})$, if $x : s$ then $s \in \mathcal{S}_0$ and $\tilde{w} : \Sigma(s)$. To allow a simple presentation of the dynamics of the calculus a relation of *structural congruence* on agent expressions, written \equiv , is defined; see [8]. The behaviour of processes is given by a family of labelled transition relations. They are obtained from the transition relations for the polyadic π -calculus by incorporating an appropriate treatment of value expressions. We adopt an ‘early’ instantiation regime [10] in which variable names of the sorts \mathbf{N} and \mathbf{B} are instantiated by constant names when an input action is inferred. The relations are labelled by actions α of which there are three kinds: the silent action τ , output actions $\bar{x}((\nu \tilde{z})\tilde{v})$ and input actions $x(\tilde{v})$. Here \tilde{v} is a tuple each of whose entries is a link name (of a sort in \mathcal{S}_0) or a constant name (of sort \mathbf{N} or \mathbf{B}), and \tilde{z} is a subset of the set of link names occurring in \tilde{v} ; \tilde{z} is the set $\text{bn}(\alpha)$ of bound names of the action α ; also $\text{bn}(\tau) = \emptyset$ and $\text{bn}(x(\tilde{v})) = \emptyset$. A *variant* of a transition $P \xrightarrow{\alpha} Q$ is a transition which differs only in that P and Q have been replaced by structurally-congruent processes and α has been alpha-converted, where a bound name of α includes Q in its scope. The relations are defined by the following rules. In each rule the conclusion stands for all variants of the transition:

1. $\bar{x}(\tilde{w}).P + \dots \xrightarrow{\bar{x}(\tilde{v})} P$ where \tilde{v} is identical to \tilde{w} except that each closed expression of sort \mathbf{N} or \mathbf{B} is replaced by its value,
2. $x(\tilde{y}).P + \dots \xrightarrow{x(\tilde{v})} P\{\tilde{v}/\tilde{y}\}$ provided the names of sort \mathbf{N} or \mathbf{B} in \tilde{v} are constant names;

here $P\{\tilde{v}/\tilde{y}\}$ is the result of substituting the components of \tilde{v} for the corresponding components of \tilde{y} in P ,

3. If $P \xrightarrow{\alpha} P'$ then $P \mid Q \xrightarrow{\alpha} P' \mid Q$ provided $\text{bn}(\alpha) \cap \text{fn}(Q) = \emptyset$,
4. If $P \xrightarrow{\bar{x}((\nu \tilde{z})\tilde{v})} P'$ and $Q \xrightarrow{x(\tilde{v})} Q'$ then $P \mid Q \xrightarrow{\tau} (\nu \tilde{z})(P' \mid Q')$ provided $\tilde{z} \cap \text{fn}(Q) = \emptyset$,
5. If $P \xrightarrow{\alpha} P'$ then $(\nu x)P \xrightarrow{\alpha} (\nu x)P'$ provided x does not occur in α ,
6. If $P \xrightarrow{\bar{x}((\nu \tilde{z})\tilde{v})} P'$ then $(\nu y)P \xrightarrow{\bar{x}((\nu \tilde{z} y)\tilde{v})} P'$ provided y occurs in $\tilde{v} - (\tilde{z} \cup \{x\})$,
7. If $P \xrightarrow{\alpha} P'$ then $\text{if}(b : P, Q) \xrightarrow{\alpha} P'$ provided the value of b is tt,
8. If $Q \xrightarrow{\alpha} Q'$ then $\text{if}(b : P, Q) \xrightarrow{\alpha} Q'$ provided the value of b is ff.

Following a standard line of development in process theory we write \Rightarrow for the reflexive and transitive closure of $\xrightarrow{\tau}$, for each α set $\xrightarrow{\alpha}$ to be $\Rightarrow \xrightarrow{\alpha} \Rightarrow$, and set $\hat{\xrightarrow{\tau}}$ to be \Rightarrow and $\hat{\xrightarrow{\alpha}}$ to be $\xrightarrow{\alpha}$ if $\alpha \neq \tau$. Then *weak bisimilarity* is the largest symmetric relation \approx on processes such that if $P \approx Q$ then for all α , if $P \xrightarrow{\alpha} P'$ then for some Q' , $Q \xrightarrow{\alpha} Q'$ and $P' \approx Q'$. We extend \approx to agent expressions by setting $P \approx Q$ if $P\sigma \approx Q\sigma$ for each σ which is the identity on link names. Similarly for abstractions, $(\tilde{y})P \approx (\tilde{y})Q$ if $P\{\tilde{v}/\tilde{y}\} \approx Q\{\tilde{v}/\tilde{y}\}$ for all \tilde{v} . The relation \approx is an equivalence relation preserved by most of the process constructors, but it is not preserved by instantiation of link names and hence not by input prefix. In carrying out the analysis of the programming examples we use the natural congruence relation on agent expressions determined by \approx . Since we work only with guarded summation no special treatment of initial τ -actions is required to characterize the congruence. We say that agent expressions P and Q are *weakly equivalent*, $P \approx Q$, if for every substitution σ , $P\sigma \approx Q\sigma$. We assume that \approx is a congruence relation on agent expressions, that it is preserved by recursive definition, and that simple equations have unique solutions up to \approx . We use also an appropriate form of expansion theorem, some straightforward algebraic laws involving conditional agent expressions, and standard τ -laws.

3.2 The Translation

To give the translation we take as subject sorts of the calculus N, B, NIL, and for each type name T and tuple \tilde{T} of type names, LINK[T], METHOD[$\tilde{T}; T$] and METHOD[$\tilde{T};$]. Here $\tilde{T}; T$ (resp. $\tilde{T};$) indicates a method taking arguments of types \tilde{T} and returning a result of type T (resp. no result type). To give the sorting it is convenient to introduce some synonyms. We set OBJECT[NAT] \equiv N and OBJECT[BOOL] \equiv B. If A has methods M_1, \dots, M_q and M_i has type $U_i = \tilde{T}_i; T_i$ or \tilde{T}_i ; then we set OBJECT[A] \equiv METHOD[U₁, ..., METHOD[U_q]]. Finally, if $\tilde{T} = T_1, \dots, T_r$ then we set OBJECTS[\tilde{T}] \equiv OBJECT[T₁, ..., OBJECT[T_r]]. The sorting Σ is then as follows:

$$\begin{aligned}\Sigma(\text{NIL}) &= () \\ \Sigma(\text{LINK}[T]) &= (\text{OBJECT}[T]) \\ \Sigma(\text{METHOD}[\tilde{T}; T]) &= (\text{OBJECTS}[\tilde{T}], \text{LINK}[T]) \\ \Sigma(\text{METHOD}[\tilde{T};]) &= (\text{OBJECTS}[\tilde{T}], \text{NIL})\end{aligned}$$

As an example, for the priority queue class Q we have

$$\begin{aligned}\text{OBJECT}[q] &\equiv \text{METHOD}[\text{NAT};], \text{METHOD}[; \text{NAT}] \\ \Sigma(\text{METHOD}[\text{NAT};]) &= (\text{N}, \text{NIL}) \\ \Sigma(\text{METHOD}[; \text{NAT}]) &= (\text{LINK}[\text{NAT}])\end{aligned}$$

Since we adopt structural matching of sorts, we have e.g. that $\text{LINK}[q] = \text{LINK}[q']$.

The translation function $\llbracket \cdot \rrbracket$ associates with each phrase of the language an agent constant whose object sort conveys some static information about the phrase. The sorting will be explained in detail as we proceed. Briefly, the ‘object identifier’ of an instance of the class Q will be a pair of names $(add, rem) : (\text{METHOD}[\text{NAT};], \text{METHOD}[\cdot; \text{NAT}])$. The invocation of the Add method in such an instance will be represented as the sending to the process encoding it, via its *add* link, of a pair $(x, r) : (N, \text{NIL})$ with x being the parameter and r a link to be used for the return of the invocation. For the Rem method, along the *rem* link is sent a return link of sort $\text{LINK}[\text{NAT}]$ for the return of the value. Channels of sort $\text{LINK}[q]$ are used to pass identifiers of objects of the class Q . The declaration of the class Q , for instance, is encoded as a replicator which may produce at a link of sort $\text{LINK}[q]$ an indefinite number of copies of the agent encoding an object of the class.

As a first illustration, abbreviating $\text{LINK}[\text{NAT}], \text{LINK}[\text{NAT}]$ to $\text{LINK}[\text{NAT}]^2$ we have

$$\llbracket \text{var } V:\text{NAT} \rrbracket : (\text{LINK}[\text{NAT}]^2) \stackrel{\text{def}}{=} (\text{get put}) \text{REG}_{\text{NAT}}(\text{get}, \text{put}, \perp)$$

where a natural number register is defined by

$$\begin{aligned} \text{REG}_{\text{NAT}} : (\text{LINK}[\text{NAT}]^2, N) &\stackrel{\text{def}}{=} (\text{get put } x) \\ &\quad (\text{get}(x). \text{REG}_{\text{NAT}}(\text{get}, \text{put}, x) + \text{put}(y). \text{REG}_{\text{NAT}}(\text{get}, \text{put}, y)) \end{aligned}$$

Here x represents the value stored, which is undefined on declaration, and *get* (*put*) the name used for reading (writing) the register. The agent $\llbracket \text{var } P:Q \rrbracket : (\text{LINK}[q]^2)$ is defined by

$$\llbracket \text{var } P:Q \rrbracket : (\text{LINK}[q]^2) \stackrel{\text{def}}{=} (\text{get put})(\nu a, r) \text{REG}_Q(\text{get}, \text{put}, a, r)$$

where $\text{REG}_Q : (\text{LINK}[q]^2, \text{OBJECT}[q])$ is similar to REG_{NAT} . The nil reference is captured by the restriction. A sequence of variable declarations is encoded as the composition of the agents representing the individual declarations.

An expression E of type T is encoded as an agent constant $\llbracket E \rrbracket : \theta^\wedge(\text{LINK}[T])$ where θ consists of sorts corresponding to the distinct variables and new expressions occurring in E , and the last parameter is used for delivering the value of the expression. For constants and variables of type NAT we set

$$\begin{aligned} \llbracket k \rrbracket : (\text{LINK}[\text{NAT}]) &\stackrel{\text{def}}{=} (\text{val}) \overline{\text{val}}(k) \\ \llbracket \text{nil} \rrbracket : (\text{LINK}[\text{NAT}]) &\stackrel{\text{def}}{=} (\text{val}) \overline{\text{val}}(\perp) \\ \llbracket x \rrbracket : (\text{LINK}[\text{NAT}]^2) &\stackrel{\text{def}}{=} (\text{get val}) \text{get}(x). \overline{\text{val}}(x) \end{aligned}$$

For the creation of a new object of class Q ,

$$\llbracket \text{new}(Q) \rrbracket : (\text{LINK}[q]^2) \stackrel{\text{def}}{=} (\text{new val}) \text{new}(a, r). \overline{\text{val}}(a, r)$$

This agent is intended to receive at *new* from the replicator representing the declaration of the class Q a pair (a, r) representing the object identifier of a new instance of the class and to yield that pair as its value at *val*. Then we have

$$\begin{aligned} \llbracket P := \text{new}(Q) \rrbracket : (\text{LINK}[q]^2, \text{NIL}) &\stackrel{\text{def}}{=} (\text{new put } d)(\nu \text{val} : \text{LINK}[q]) \\ &\quad (\llbracket \text{new}(Q) \rrbracket(\text{new}, \text{val}) \mid \text{val}(a, r). \overline{\text{put}}(a, r). \bar{d}) \end{aligned}$$

Here *put* is a link to the register agent corresponding to the variable P and d a link on which the agent encoding the assignment statement signals termination. In general, a statement S is encoded as an agent $\llbracket S \rrbracket : \theta^\wedge(\text{NIL})$ where θ consists of sorts corresponding to the distinct variables and new expressions occurring in S together with names associated with method calls.

Continuing the illustration we have $\llbracket P!Add(X) \rrbracket : (\text{LINK}[q], \text{LINK}[\text{NAT}], \text{NIL})$ defined by

$$\begin{aligned} \llbracket P!Add(X) \rrbracket &\stackrel{\text{def}}{=} (get_P get_X d) \quad (\nu val_0 : \text{LINK[q]}, val_1 : \text{LINK[NAT]}, w : \text{NIL}) \\ &\quad (\llbracket P \rrbracket \langle get_P, val_0 \rangle \mid w. \llbracket X \rrbracket \langle get_X, val_1 \rangle) \\ &\quad | val_0(a, r). \overline{w}. val_1(y). (\nu ret) \overline{a}(y, ret). ret. \overline{d} \end{aligned}$$

The third component receives the pair of links representing the reference stored in P , activates the second component and receives from it the value of the parameter, then sends that with a private return link along the Add -component of the reference. It then waits for a return along ret before signalling termination at d . The boolean expression $V<X$ is encoded as an agent of sort $(\text{LINK[NAT]}^2, \text{LINK[BOOL]})$ thus:

$$\begin{aligned} \llbracket V<X \rrbracket &\stackrel{\text{def}}{=} (get_V get_X val) \quad (\nu val_0 : \text{LINK[NAT]}, val_1 : \text{LINK[NAT]}, w : \text{NIL}) \\ &\quad (\llbracket V \rrbracket \langle get_V, val_0 \rangle \mid w. \llbracket X \rrbracket \langle get_X, val_1 \rangle \mid \text{LESS}\langle val_0, val_1, w, val \rangle) \end{aligned}$$

where $\text{LESS} : (\text{LINK[NAT]}^2, \text{NIL}, \text{LINK[BOOL]})$ is defined by

$$\text{LESS} \stackrel{\text{def}}{=} (val_0 val_1 w val) val_0(x_0). \overline{w}. val_1(x_1). \overline{val}(x_0 < x_1)$$

Continuing further, setting $\theta = (\text{LINK[NAT]}^2, \text{LINK[q]}, \text{LINK[NAT]}, \text{NIL})$ we have

$$\begin{aligned} \llbracket \text{if } V<X \text{ then } P!Add(X) \text{ else } (P!Add(V) ; V:=X) \rrbracket : \theta &\stackrel{\text{def}}{=} \\ (get_V put_V get_P get_X d) \quad (\nu val : \text{LINK[BOOL]}, d_0 : \text{NIL}, d_1 : \text{NIL}) & \\ (\llbracket V<X \rrbracket \langle get_V, get_X, val \rangle & \\ | val(b). \text{if}(b : d_0, d_1) & \\ | d_0. \llbracket P!Add(X) \rrbracket \langle get_P, get_X, d \rangle & \\ | d_1. \llbracket P!Add(V) ; V:=X \rrbracket \langle get_P, get_V, put_V, get_X, d \rangle) & \end{aligned}$$

Continuing with the illustration we have that for the body S_{Add} of the Add method of Q , $S_{\text{Add}} : (\text{LINK[q]}, \text{LINK[NAT]}^2, \text{LINK[q]}^2, \text{LINK[NAT]}, \text{NIL}^2)$ where the penultimate abstracted name is used in encoding the return statement in S_{Add} . We then have $\llbracket \text{method Add}(X:\text{NAT}), S_{\text{Add}} \rrbracket : (\text{LINK[q]}, \text{LINK[NAT]}^2, \text{LINK[q]}^2, \text{METHOD[NAT]}, \text{NIL})$ defined by

$$\begin{aligned} \llbracket \text{method Add}(X:\text{NAT}), S_{\text{Add}} \rrbracket &\stackrel{\text{def}}{=} (new get_V put_V get_P put_P add d) \\ (\nu get_X, put_X) (\llbracket \text{var } X:\text{NAT} \rrbracket \langle get_X, put_X \rangle & \\ | add(y, ret). put_X(y). \llbracket S_{\text{Add}} \rrbracket \langle new, get_V, put_V, get_P, put_P, get_X, ret, d \rangle) & \end{aligned}$$

The encoding of the Rem method is along similar lines. Then an instance of class Q in the quiescent state is represented by the agent $\text{Obj}_Q : (\text{LINK[q]}, \text{OBJECT[q]})$ defined as follows where $\tilde{p} = get_V, put_V, get_P, put_P$:

$$\text{Obj}_Q \stackrel{\text{def}}{=} (new add rem)(\nu \tilde{p})(\llbracket \text{var } V:\text{NAT}, P:Q \rrbracket \langle \tilde{p} \rangle \mid \text{Body}_Q \langle new, \tilde{p}, add, rem \rangle)$$

where

$$\text{Body}_Q \stackrel{\text{def}}{=} (new add rem)(\nu d) ((\llbracket \text{method Add} \rrbracket \langle \dots, add, d \rangle + \llbracket \text{method Rem} \rrbracket \langle \dots, rem, d \rangle) \\ | d. \text{Body}_Q \langle new, add, rem \rangle)$$

This captures that in the quiescent state the object offers its two methods, and that it returns to the quiescent state when execution of a method invocation finishes. As mentioned earlier the declaration of the class Q is encoded as a replicator:

$$\llbracket \text{class } Q \rrbracket \stackrel{\text{def}}{=} (new)(\nu add, rem)! \overline{new}(add, rem). \text{Obj}_Q \langle new, add, rem \rangle$$

Here ‘!’ is the replication operator from [8] which may be eliminated in favour of an agent constant. We can think of $!P$ as $P \mid P \mid P \mid \dots$. Finally the encoding of a program is obtained by composing the representations of the class declarations with the agent corresponding to the root object, and restricting on all link names except those which correspond to the interface of the program.

4 Priority Queues

In this section we describe the analysis of the priority queue classes Q and Q' outlined in the Introduction. Due to the limitation of space, very many details are omitted. Suppose prog is a program in which the definition of Q appears. Then as outlined in the preceding section its encoding $\llbracket \text{prog} \rrbracket$ is an abstraction of the form

$$(\dots) (\nu \dots) (\text{Classes}\{\dots\} \mid \text{RootObject}\{\dots\})$$

where Classes is the composition of the replicators which represent the class definitions and RootObject is an agent representing the root object of the system. $\llbracket \text{prog} \rrbracket$ is abstracted on the link names which correspond to the interface of the program; all other link names are restricted. Consider the context

$$\mathcal{C}[\bullet] \equiv (\dots) (\nu \text{new}, \dots) (\bullet \langle \text{new} \rangle \mid \text{OtherClasses}\{\dots\} \mid \text{RootObject}\{\dots\})$$

derived by encoding the incomplete program obtained by omitting the definition of Q ; the ‘•’ indicates where the abstraction $\llbracket \text{class } Q \rrbracket$ should be placed to recover $\llbracket \text{prog} \rrbracket$. We assume that the name new on which the encoding of $\llbracket \text{class } Q \rrbracket$ is abstracted is not part of the program’s interface and is thus restricted. Let prog' be obtained from prog by replacing the definition of Q with that of Q' and each mention of Q by mention of Q' . The behavioural equivalence of prog and prog' may be expressed as follows:

Theorem 1 $\mathcal{C}[\llbracket \text{class } Q \rrbracket] \approx \mathcal{C}[\llbracket \text{class } Q' \rrbracket]$.

A sketch of the proof follows. If the root object is an instance of one of the priority queue classes then since the only class new instances of which may be created by the root object is the class of which it is an instance, the result follows easily. To prove the theorem in the more interesting case, two principal results are required. The first is an addition to the collection of useful properties of replicators presented in [8].

Theorem 2 Suppose that $R \equiv !(\nu \tilde{y})\bar{x}(\tilde{y}).P$, $R' \equiv !(\nu \tilde{y})\bar{x}(\tilde{y}).P'$ and $(\nu x)(R \mid P) \approx (\nu x)(R' \mid P')$ where x occurs in P and P' only in positive subject position. Then for any S in which x occurs only in positive subject position, $(\nu x)(R \mid S) \approx (\nu x)(R' \mid S)$.

Proof: Assume $(\nu x)(R \mid P) \approx (\nu x)(R' \mid P')$ where R, P, R', P' are as above. Set $E \mathcal{B} E'$ if $E \equiv (\nu x \tilde{z})(R \mid S)$ and $E' \equiv (\nu x \tilde{z})(R' \mid S')$ for some \tilde{z} and $S \approx S'$ in both of which x occurs only in positive subject position. We claim that \mathcal{B} is a weak bisimulation up to strong bisimilarity \sim . To prove this we first recall from [8] that $(\nu x)(R \mid P) \sim (\nu x)(R \mid (\nu x)(R \mid P))$ and similarly for R' where \sim is strong equivalence. Suppose $E \mathcal{B} E'$ where E and E' are as above and $E \xrightarrow{\alpha} F$. Then either $F \equiv (\nu x \tilde{w})(R \mid T)$ where $S \xrightarrow{\alpha} T$, or $F \equiv (\nu x \tilde{z} \tilde{y})(R \mid P \mid T)$ where $S \xrightarrow{x(\tilde{y})} T$. In the first case $S' \xrightarrow{\hat{\alpha}} T' \approx T$ so $E' \xrightarrow{\hat{\alpha}} F' \equiv (\nu x \tilde{w})(R' \mid T')$ and $F \mathcal{B} F'$. In the second case $S' \xrightarrow{\#(\tilde{y})} T' \approx T$ so $E' \xrightarrow{\#} F' \equiv (\nu x \tilde{z} \tilde{y})(R' \mid P' \mid T')$. Now by the fact noted above, $F \sim G \equiv (\nu x \tilde{z} \tilde{y})(R \mid (\nu x)(R \mid P) \mid T)$ and $F' \sim G' \equiv (\nu x \tilde{z} \tilde{y})(R' \mid (\nu x)(R' \mid P') \mid T')$. Moreover as $(\nu x)(R \mid P) \approx (\nu x)(R' \mid P')$ and $T \approx T'$, $G \mathcal{B} G'$. Hence $E' \xrightarrow{\#} F'$ with $F \sim \mathcal{B} \sim F'$. So \mathcal{B} is a weak bisimulation up to \sim and hence if x occurs in S only in positive subject position, $(\nu x)(R \mid S) \approx (\nu x)(R' \mid S)$. Now if σ is a substitution in which x does not occur then repeating the above argument replacing R, P, R', P' by $R\sigma, P\sigma, R'\sigma, P'\sigma$ we have that if x occurs in T only in positive subject position, $(\nu x)(R\sigma \mid T) \approx (\nu x)(R'\sigma \mid T)$. Hence if x occurs in S only in positive subject position then for any σ , $((\nu x)(R \mid S))\sigma \equiv (\nu x)(R\sigma' \mid S\sigma') \approx (\nu x)(R'\sigma' \mid S\sigma') \equiv ((\nu x)(R' \mid S))\sigma$ for some σ' in which x does not occur. Hence $(\nu x)(R \mid S) \approx (\nu x)(R' \mid S)$. \square

The second result describes a close relationship between the agents encoding the classes.

Theorem 3 We have

$$(\nu \text{ new})([\![\text{class Q}]\!]\langle \text{new} \rangle \mid \text{Obj}_q\langle \text{new}, \dots \rangle) \approx (\nu \text{ new})([\![\text{class Q}']\!]\langle \text{new} \rangle \mid \text{Obj}'_q\langle \text{new}, \dots \rangle).$$

To prove Theorem 1, in Theorem 2 we take $R \equiv [\![\text{class Q}]\!]\langle \text{new} \rangle$, $R' \equiv [\![\text{class Q}']\!]\langle \text{new} \rangle$ and S the context $\mathcal{C}[\bullet]$ without the restriction on new so that $(\nu \text{ new})(R \mid S) \equiv \mathcal{C}[\![\text{class Q}]\!]$ and $(\nu \text{ new})(R' \mid S) \equiv \mathcal{C}[\![\text{class Q}']\!]$. The condition that new occur in Obj_q , Obj'_q and S only in positive subject position is met: it is straightforward to check from the full definition of the translation $[\cdot]$. Furthermore, by Theorem 3 the other condition of Theorem 2 is met in this case. Hence an appeal to Theorem 2 gives the desired result, that $\mathcal{C}[\![\text{class Q}]\!] \approx \mathcal{C}[\![\text{class Q}']\!]$.

To prove Theorem 3 is less straightforward than it might appear at first glance. It is fairly easy to *describe* a relation containing the relevant pair of agents which one would expect to be a bisimulation up to \approx . But to *prove* that it is so would require an argument taking account of the fact that an indefinite number of new cell-agents may be created, and that each time a new cell is added to the chain, new behaviour becomes possible. To overcome this difficulty we distil from an understanding of the abstract behaviour of the classes a system of equations in agent variables which by virtue of its form is known to have a unique solution up to \approx . We then derive abstract descriptions of the behaviour of the agents Obj_q and Obj'_q and use these to give inductive proofs that the processes in Theorem 3 are corresponding members of two families which are solutions to this system. See [16] for the proofs and an account of the treatment of run-time errors, an issue which must be addressed in order for the above discussion to be fully accurate. This topic is addressed briefly in the case of the symbol tables at the beginning of the next section.

5 Symbol Tables

In this section we outline the analysis of the relationship between the symbol table classes T and T' described in the Introduction. As explained there the correspondence between the classes is not a simple one. In particular there is no simple behavioural equivalence between the agents encoding them. Thus the analysis differs markedly from that for the priority queue classes. As mentioned at the end of the previous section we must take account of the possibility of run-time errors. We do this by introducing an empty statement λ and inserting an extra conditional at the head of each method body of class T :

```

method Insert(X:NAT, W:A)
  if X=nil then λ
  else if K=nil then ...
  
method Search(X:NAT)
  if X=nil then λ
  else if K=nil then ...

```

For the *Search* method of class T' the alteration is identical to that above while for the *Insert* method we have

```

method Insert(X:NAT, W:A)
  return ;
  if X=nil then λ
  else if K=nil then ...

```

The treatment of run-time errors in the priority queue example is similar. The reason for making these modifications to the class definitions is that in their absence, due to the possibility of a method being invoked with a parameter having an undefined value, the precise relationship between the class definitions would be much less simple. When, for instance, the `Insert` method of a T' -node is invoked, the caller is released from the rendezvous before the parameter is examined. In contrast, if a T -node examines the parameter, it does so before returning to the caller. Since, according to the semantics by translation, the process encoding a node may deadlock during examination of an undefined value – an agent of the form $\text{if}(\perp : P, Q)$ has no transitions – it is clear that as a tree structure evolves, significantly different behaviour may be manifest in the two cases. A related issue is the treatment of searches for keys absent from a table. We assume that the result of such a search is a nil reference and use the expression $\tilde{r}(\text{nil})$ to represent the return on a link r of a tuple of names corresponding to a nil reference.

We first obtain an abstract description of the behaviour generated by the sequential class T . To describe symbol tables we use the tree expressions given by

$$t ::= \varepsilon \mid t_l \triangleleft \langle k, a \rangle \triangleright t_r$$

Here ε represents a tree consisting of a single node with key \perp , and in the tree described by $t_l \triangleleft \langle k, a \rangle \triangleright t_r$, the root has key k and value a , if h is a key of the left subtree t_l then $h < k$, and if h is a key of the right subtree t_r , then $h > k$. We write \tilde{t}_x for the value associated with key x in t (if it exists).

To represent trees of nodes of the class T we introduce agents T_t as follows. Recalling that $\text{Obj}_T : (\text{LINK}[T], \text{OBJECT}[T])$ where $\text{OBJECT}[T] \equiv \text{METHOD}[\text{NAT}, A]$, $\text{METHOD}[\text{NAT}; A]$ we have $T_t : (\text{LINK}[T], \text{OBJECT}[T], (N, \text{OBJECT}[A])^h, N^l)$ if t represents a tree with l leaves and h internal nodes, where the linear order of the parameters is related in some natural way to the corresponding tree structure. Let $E \equiv \text{Obj}_T(new, ins, srch)$ be the process representing an empty node of class T . Let also $C\langle k, a \rangle \equiv C\langle new, ins, srch, k, a, insl, srchl, insr, srchr \rangle$ be the representation of a node of class T storing an integer key k and with $a : \text{OBJECT}[A]$ representing the reference stored in the variable V and $(insl, srchl)$, resp. $(insr, srchr)$, the reference stored in the variable L , resp. R . Then $T_\varepsilon(\tilde{p}, \perp) \equiv E$ where $\tilde{p} = new, ins, srch$, and for $t = t_l \triangleleft \langle x, a \rangle \triangleright t_r$,

$$T_t \stackrel{\text{def}}{=} (\tilde{p} x a \dots)(\nu \tilde{q})(T_{t_l}(\tilde{p}_l, \dots) \mid C\langle x, a \rangle \mid T_{t_r}(\tilde{p}_r, \dots))$$

where $\tilde{q} = insl, srchl, insr, srchr$, $\tilde{p}_l = new, insl, srchl$ and $\tilde{p}_r = new, insr, srchr$. We define also $T_t^\sim : (N, \text{OBJECT}[A], \text{LINK}[T], \text{OBJECT}[T], (N, \text{OBJECT}[A])^h, N^l)$ as follows:

$$T_t^\sim \stackrel{\text{def}}{=} (y b \tilde{p} \dots) \text{if}(y = \perp : T_\varepsilon(\tilde{p}, \perp), T_{t_0}(\tilde{p}, y, b, \perp, \perp))$$

where $t_0 = \varepsilon \triangleleft \langle y, b \rangle \triangleright \varepsilon$, and for $t = t_l \triangleleft \langle x, a \rangle \triangleright t_r$,

$$\begin{aligned} T_t^\sim \stackrel{\text{def}}{=} & (y b \tilde{p} \dots) \text{cond}(y = \perp : T_t(\tilde{p}, x, a, \dots), \\ & y = x : T_{t'}(\tilde{p}, x, b, \dots), \\ & y < x : (\nu \tilde{q})(T_{t_l}^\sim \langle y, b, \tilde{p}_l, \dots \rangle \mid C\langle x, a \rangle \mid T_{t_r}(\tilde{p}_r, \dots)), \\ & \text{else} : (\nu \tilde{q})(T_{t_l}(\tilde{p}_l, \dots) \mid C\langle x, a \rangle \mid T_{t_r}^\sim \langle y, b, \tilde{p}_r, \dots \rangle)) \end{aligned}$$

where $t' = t_l \triangleleft \langle x, b \rangle \triangleright t_r$ and `cond` abbreviates a nested conditional. Finally define

$$\tilde{E} \equiv (\nu new)(E \mid [\text{class } T](new))$$

and define $\tilde{C}\langle k, a \rangle$, \tilde{T}_t and \tilde{T}_t^\sim similarly as restricted compositions. Then omitting some parameters to improve readability we have the following.

Theorem 4 For any t , $\tilde{T}_t(\dots) \approx \text{ins}(y, b, r). \bar{r}. \tilde{T}_t^\sim(y, b, \dots) + \text{srch}(x, r). \bar{r}(\hat{x}). \tilde{T}_t(\dots)$.

This result is proved by finding abstract descriptions of \tilde{E} and \tilde{C} and then establishing, using an inductive argument, that the family $\{\tilde{T}_t, \tilde{T}_t^\sim\}_t$ satisfies the system of equations implicit in the theorem. Note that $\tilde{T}_e(\tilde{p}, \perp) \equiv (\nu \text{ new})(\text{Obj}_T(\tilde{p}) \mid [\text{class } T](\text{new}))$ so the theorem gives a convenient description of the behaviour of a newly-created object of class T (in restricted correspondence with a private copy of the replicator representing the class). We now wish to give a corresponding abstract description of the behaviour of $(\nu \text{ new})(\text{Obj}'_T(\tilde{p}) \mid [\text{class } T'](\text{new}))$.

To describe the behaviour of a tree of T' -nodes we consider augmented tree expressions (t, σ) where t is a tree expression and σ a function which assigns to each node n of the tree described by t a *status* $\sigma(n)$. If n is a leaf then $\sigma(n)$ may be f or of the form (i_r, m, b, r) or the form (s, m, r) . The first indicates that the node is inactive ('free'), the second that the *Insert* method with parameters (m, b) and return link r has been invoked but has not yet returned, and the third that the *Search* method has been invoked with parameter m and return link r but that the search has not yet been returned or committed. If n is an internal node with key k then $\sigma(n)$ may additionally be of the form (i, m, b) where $m \neq \perp, k$. This indicates that the *Insert* method with parameters (m, b) has been invoked and returned but not yet passed on to the appropriate child node.

The abstract description of the behaviour of trees of T' -nodes is obtained by focussing on augmented tree expressions (t, σ) in which each method invocation has progressed as far as it can through the tree structure, subject to the constraints imposed by other outstanding invocations. To do this we use the function *fall* on augmented tree expressions which is such that $\text{fall}(t, \sigma)$ describes a tree in which the method invocation outstanding at the root node, if there is one, is propagated as far through the tree structure as possible without any other method invocation making further progress.

For n a node of the tree described by an augmented tree expression (t, σ) we let $\text{fall}(t, \sigma, n)$ be the augmented tree expression describing the tree obtained by replacing the subtree (t_n, σ_n) rooted at n by $\text{fall}(t_n, \sigma_n)$. Now suppose n is a node of the tree described by (t, σ) and let $n = n_1, \dots, n_p = \text{root}(t)$ be the nodes on the path from n to the root of t . Define $(t_1, \sigma_1) = \text{fall}(t, \sigma, n_1)$ and for $2 \leq i \leq p$, $(t_i, \sigma_i) = \text{fall}(t_{i-1}, \sigma_{i-1}, n_i)$. Then define $\text{cascade}(t, \sigma, n) = (t_p, \sigma_p)$. The function *cascade* is useful in describing the overall progress of method invocations when a search is returned by a node. We then say that (t, σ) is *settled* if for each node n of t , $\text{cascade}(t, \sigma, n) = (t, \sigma)$. In giving the abstract description we stay within the domain of settled expressions. We say also that a node n of t is *ready* (to return a search) if for some m and r , $\sigma(n) = (s, m, r)$ and n has key m or key \perp . Finally we say (t, σ) is *blocked* if $\sigma(\text{root}(t)) \neq f$.

We can now define the agents which provide an abstract description of the behaviour of trees of T' -nodes. Suppose (t, σ) is settled. If (t, σ) is blocked we set $U_{t, \sigma} \stackrel{\text{def}}{=} U_{t, \sigma}^R$ while if (t, σ) is not blocked we set

$$U_{t, \sigma} \stackrel{\text{def}}{=} U_{t, \sigma}^R + \text{ins}(y, b, r). U_{t', \sigma'} + \text{srch}(x, r). U_{t'', \sigma''}$$

where $(t', \sigma') = (t, \sigma[(i_r, y, b, r)/\text{root}(t)])$ and $(t'', \sigma'') = \text{fall}(t, \sigma[(s, x, r)/\text{root}(t)])$, and where

$$U_{t, \sigma}^R \stackrel{\text{def}}{=} \Sigma \{ \bar{r}(a). U_{t', \sigma'} \mid \text{for some ready node } n \text{ of } t, t(n) = \langle k, a \rangle \text{ and } \sigma(n) = (s, k, r), \\ \text{and } (t', \sigma') = \text{cascade}(t, \sigma[f/n], n) \}$$

$$+ \Sigma \{ \bar{r}(\text{nil}). U_{t', \sigma'} \mid \text{for some leaf } n \text{ of } t, \sigma(n) = (s, k, r) \text{ and} \\ (t', \sigma') = \text{cascade}(t, \sigma[f/n], n) \}$$

$$+ \bar{r}. U_{t', \sigma'}$$

The last summand is present only if the root node has status (i_r, m, b, r) , so that (t, σ) is blocked, and in it (t', σ') is the appropriate expression representing the *fall* of the insertion as far as possible through the tree.

The agent $U_{t, \sigma}^R$ describes the possible returns of method invocations from the tree of T' -nodes described by (t, σ) . The first summand describes the returns of searches for keys present in the table, the second the return of searches for keys absent from the table, and the third the return from the root node of an `Insert` method. That this family of agents describes the behaviour of trees of T' -nodes is the substance of the following theorem.

Theorem 5 Setting $(t_0, \sigma_0) = (\varepsilon, [f/root])$ we have

$$(\nu \text{ new})(\text{Obj}_T(\text{new}, \text{ins}, \text{srch}) \mid [\text{class } T](\text{new})) \approx U_{t_0, \sigma_0}.$$

The proof is again by induction and involves finding abstract descriptions of the behaviour of agents encoding nodes. Using these abstract descriptions of the behaviour of $(\nu \text{ new})(\text{Obj}_T(\dots) \mid [\text{class } T](\text{new}))$ and $(\nu \text{ new})(\text{Obj}_{T'}(\dots) \mid [\text{class } T'](new))$, we may now describe the relationship between the classes sketched in the Introduction. Let $\tilde{\mu} = \mu_1, \dots, \mu_p$ be any sequence of method invocations in which none of the parameters is \perp and which is such that if μ_i is `Search(k)` then for some $j < i$, μ_j is `Insert(k, a)` for some reference a . (It is not essential to make these restrictions but doing so allows us to concentrate on the main case). Associated with $\tilde{\mu}$ are a sequence $\tilde{\alpha} = \alpha_1, \dots, \alpha_{2p}$ of actions and a sequence $\tilde{t} = t_0, \dots, t_p$ of tree expressions, where $t_0 = \varepsilon$, if μ_i is `Insert(k, a)` then for some r , $\alpha_{2i-1} = \text{ins}(k, a, r)$ where a corresponds to a , $\alpha_{2i} = \bar{r}$, and $\tilde{t}_i = \widehat{t_{i-1}[a/k]}$, and if μ_i is `Search(k)` then for some r , $\alpha_{2i-1} = \text{srch}(k, r)$, $\alpha_{2i} = \bar{r}(\widehat{t_{i-1}k})$ and $t_i = t_{i-1}$.

By Theorem 4, $(\nu \text{ new})(\text{Obj}_T(\dots) \mid [\text{class } T](\text{new})) \xrightarrow{\tilde{\alpha}} \approx \tilde{T}_{t_p}(\dots)$ is the unique computation of the T -node (with private replicator) determined by $\tilde{\mu}$. Theorem 5 describes the possible computations of the T' -node (with private replicator) determined by $\tilde{\mu}$. They are labelled by the permutations of the sequence $\tilde{\alpha}$ allowed by the evolving tree structure. Note that in the encoding of a program in which T' occurs, whenever an object-agent invokes a search in a T' -table it creates a private link for the return of the result and suspends its activity until a result is returned to it. This private name is handled (in the precise sense of [8]) by only one T' -node agent at any one time. Moreover the restriction operator ensures that the return links associated with distinct searches are not confused and that the result of a search is returned to the appropriate invoking object.

6 Conclusion

The process calculus apparatus employed to analyse the examples has been limited to purely equational reasoning. This approach is certainly appropriate for establishing the equivalence of the encodings of the priority queue classes and there are no doubt other instances of equivalence-preserving transformations whose soundness could be established by analogous reasoning. In the symbol tables example the equational techniques are useful in giving abstract descriptions of the behaviour generated by the two classes. From these the precise, intricate relationship between the classes may be seen more clearly. Remaining within the process calculus framework, one might also investigate the modal and temporal properties of the agents encoding the classes, and examine whether the use of a higher-order calculus would ease the analysis. It would also be of interest to examine alternative approaches to the question of the soundness of the kind of program transformations considered here. For instance, can the metric-space semantics of [2] be used to tackle this kind of question? And can the Hoare-style proof system for partial correctness of POOL programs presented in [4], or extensions of it, be used fruitfully in studying such problems? This paper has addressed

the issue of the soundness of program transformations through two concrete examples. It is hoped that it may contribute to the development of techniques for proving soundness of general transformation rules for parallel object-oriented programs.

References

- [1] P. America, J. de Bakker, J. Kok and J. Rutten, Operational semantics of a parallel object-oriented language, in *Conference Record of the 13th Symposium on Principles of Programming Languages*, 194–208 (1986).
- [2] P. America, J. de Bakker, J. Kok and J. Rutten, Denotational semantics of a parallel object-oriented language, *Information and Computation*, 83, 152–205 (1989).
- [3] P. America, Issues in the design of a parallel object-oriented language, *Formal Aspects of Computing*, 1, 366–411 (1989).
- [4] F. de Boer, Reasoning about dynamically evolving process structures, PhD thesis, Free University of Amsterdam (1991).
- [5] C. Jones, Constraining interference in an object-based design method, in Proc. TAP-SOFT'93, Springer-Verlag LNCS vol. 668, 136–150 (1993).
- [6] C. Jones, A pi-calculus semantics for an object-based design notation, in Proc. CONCUR'93, Springer-Verlag LNCS vol. 715, 158–172 (1993).
- [7] R. Milner, **Communication and Concurrency**, Prentice Hall, 1989.
- [8] R. Milner, The polyadic π -calculus: a tutorial, in **Logic and Algebra of Specification**, F. Bauer et al. (eds.), Springer-Verlag (1993).
- [9] R. Milner, J. Parrow and D. Walker, A calculus of mobile processes, I and II, *Information and Computation* 100, 1–40 and 41–77 (1992).
- [10] R. Milner, J. Parrow and D. Walker, Modal logics for mobile processes, in Proc. CONCUR'91, Springer-Verlag LNCS vol. 527, 45–60 (1991).
- [11] B. Pierce and D. Sangiorgi, Typing and subtyping for mobile processes, in Proc. IEEE LICS'93, 376–385, Computer Society Press (1993).
- [12] D. Sangiorgi, Expressing mobility in process algebras: first-order and higher-order paradigms, PhD thesis, University of Edinburgh (1992).
- [13] D. Walker, π -calculus semantics for object-oriented programming languages, in Proc. TACS'91, Springer-Verlag LNCS 526, 532–547 (1991).
- [14] D. Walker, Objects in the π -calculus, to appear in *Information and Computation* (1992).
- [15] D. Walker, Process calculus and parallel object-oriented programming lanaguges, in Proc. International Summer Institute on Parallel Computer Architectures, Langauges and Algorithms, Prague, July 1993 (Computer Society Press, to appear).
- [16] D. Walker, Algebraic proofs of properties of objects, technical report, University of Warwick (1994).

Lecture Notes in Computer Science

For information about Vols. 1–709
please contact your bookseller or Springer-Verlag

- Vol. 710: Z. Ésik (Ed.), *Fundamentals of Computation Theory*. Proceedings, 1993. IX, 471 pages. 1993.
- Vol. 711: A. M. Borzyszkowski, S. Sokółowski (Eds.), *Mathematical Foundations of Computer Science 1993*. Proceedings, 1993. XIII, 782 pages. 1993.
- Vol. 712: P. V. Rangan (Ed.), *Network and Operating System Support for Digital Audio and Video*. Proceedings, 1992. X, 416 pages. 1993.
- Vol. 713: G. Gottlob, A. Leitsch, D. Mundici (Eds.), *Computational Logic and Proof Theory*. Proceedings, 1993. XI, 348 pages. 1993.
- Vol. 714: M. Bruynooghe, J. Penjam (Eds.), *Programming Language Implementation and Logic Programming*. Proceedings, 1993. XI, 421 pages. 1993.
- Vol. 715: E. Best (Ed.), *CONCUR'93*. Proceedings, 1993. IX, 541 pages. 1993.
- Vol. 716: A. U. Frank, I. Campari (Eds.), *Spatial Information Theory*. Proceedings, 1993. XI, 478 pages. 1993.
- Vol. 717: I. Sommerville, M. Paul (Eds.), *Software Engineering – ESEC '93*. Proceedings, 1993. XII, 516 pages. 1993.
- Vol. 718: J. Seberry, Y. Zheng (Eds.), *Advances in Cryptology – AUSCRYPT '92*. Proceedings, 1992. XIII, 543 pages. 1993.
- Vol. 719: D. Chetverikov, W.G. Kropatsch (Eds.), *Computer Analysis of Images and Patterns*. Proceedings, 1993. XVI, 857 pages. 1993.
- Vol. 720: V. Mařík, J. Lažanský, R.R. Wagner (Eds.), *Database and Expert Systems Applications*. Proceedings, 1993. XV, 768 pages. 1993.
- Vol. 721: J. Fitch (Ed.), *Design and Implementation of Symbolic Computation Systems*. Proceedings, 1992. VIII, 215 pages. 1993.
- Vol. 722: A. Miola (Ed.), *Design and Implementation of Symbolic Computation Systems*. Proceedings, 1993. XII, 384 pages. 1993.
- Vol. 723: N. Aussenac, G. Boy, B. Gaines, M. Linster, J.-G. Ganascia, Y. Kodratoff (Eds.), *Knowledge Acquisition for Knowledge-Based Systems*. Proceedings, 1993. XIII, 446 pages. 1993. (Subseries LNAI)
- Vol. 724: P. Cousot, M. Falaschi, G. Filè, A. Rauzy (Eds.), *Static Analysis*. Proceedings, 1993. IX, 283 pages. 1993.
- Vol. 725: A. Schiper (Ed.), *Distributed Algorithms*. Proceedings, 1993. VIII, 325 pages. 1993.
- Vol. 726: T. Lengauer (Ed.), *Algorithms – ESA '93*. Proceedings, 1993. IX, 419 pages. 1993.
- Vol. 727: M. Filgueiras, L. Damas (Eds.), *Progress in Artificial Intelligence*. Proceedings, 1993. X, 362 pages. 1993. (Subseries LNAI).
- Vol. 728: P. Torasso (Ed.), *Advances in Artificial Intelligence*. Proceedings, 1993. XI, 336 pages. 1993. (Subseries LNAI).
- Vol. 729: L. Donatiello, R. Nelson (Eds.), *Performance Evaluation of Computer and Communication Systems*. Proceedings, 1993. VIII, 675 pages. 1993.
- Vol. 730: D. B. Lomet (Ed.), *Foundations of Data Organization and Algorithms*. Proceedings, 1993. XII, 412 pages. 1993.
- Vol. 731: A. Schill (Ed.), *DCE – The OSF Distributed Computing Environment*. Proceedings, 1993. VIII, 285 pages. 1993.
- Vol. 732: A. Bode, M. Dal Cin (Eds.), *Parallel Computer Architectures*, IX, 311 pages. 1993.
- Vol. 733: Th. Grechenig, M. Tscheiligi (Eds.), *Human Computer Interaction*. Proceedings, 1993. XIV, 450 pages. 1993.
- Vol. 734: J. Volkert (Ed.), *Parallel Computation*. Proceedings, 1993. VIII, 248 pages. 1993.
- Vol. 735: D. Bjørner, M. Broy, I. V. Pottosin (Eds.), *Formal Methods in Programming and Their Applications*. Proceedings, 1993. IX, 434 pages. 1993.
- Vol. 736: R. L. Grossman, A. Nerode, A. P. Ravn, H. Rischel (Eds.), *Hybrid Systems*, VIII, 474 pages. 1993.
- Vol. 737: J. Calmet, J. A. Campbell (Eds.), *Artificial Intelligence and Symbolic Mathematical Computing*. Proceedings, 1992. VIII, 305 pages. 1993.
- Vol. 738: M. Weber, M. Simons, Ch. Lafontaine, *The Generic Development Language Deva*, XI, 246 pages. 1993.
- Vol. 739: H. Imai, R. L. Rivest, T. Matsumoto (Eds.), *Advances in Cryptology – ASIACRYPT '91*. X, 499 pages. 1993.
- Vol. 740: E. F. Brickell (Ed.), *Advances in Cryptology – CRYPTO '92*. Proceedings, 1992. X, 593 pages. 1993.
- Vol. 741: B. Preneel, R. Govaerts, J. Vandewalle (Eds.), *Computer Security and Industrial Cryptography*. Proceedings, 1991. VIII, 275 pages. 1993.
- Vol. 742: S. Nishio, A. Yonezawa (Eds.), *Object Technologies for Advanced Software*. Proceedings, 1993. X, 543 pages. 1993.
- Vol. 743: S. Doshita, K. Furukawa, K. P. Jantke, T. Nishida (Eds.), *Algorithmic Learning Theory*. Proceedings, 1992. X, 260 pages. 1993. (Subseries LNAI)
- Vol. 744: K. P. Jantke, T. Yokomori, S. Kobayashi, E. Tomita (Eds.), *Algorithmic Learning Theory*. Proceedings, 1993. XI, 423 pages. 1993. (Subseries LNAI)
- Vol. 745: V. Roberto (Ed.), *Intelligent Perceptual Systems*, VIII, 378 pages. 1993. (Subseries LNAI)

- Vol. 746: A. S. Tanguiane, Artificial Perception and Music Recognition. XV, 210 pages. 1993. (Subseries LNAI).
- Vol. 747: M. Clarke, R. Kruse, S. Moral (Eds.), Symbolic and Quantitative Approaches to Reasoning and Uncertainty. Proceedings, 1993. X, 390 pages. 1993.
- Vol. 748: R. H. Halstead Jr., T. Ito (Eds.), Parallel Symbolic Computing: Languages, Systems, and Applications. Proceedings, 1992. X, 419 pages. 1993.
- Vol. 749: P. A. Fritzson (Ed.), Automated and Algorithmic Debugging. Proceedings, 1993. VIII, 369 pages. 1993.
- Vol. 750: J. L. Díaz-Herrera (Ed.), Software Engineering Education. Proceedings, 1994. XII, 601 pages. 1994.
- Vol. 751: B. Jähne, Spatio-Temporal Image Processing. XII, 208 pages. 1993.
- Vol. 752: T. W. Finin, C. K. Nicholas, Y. Yesha (Eds.), Information and Knowledge Management. Proceedings, 1992. VII, 142 pages. 1993.
- Vol. 753: L. J. Bass, J. Gornostaev, C. Unger (Eds.), Human-Computer Interaction. Proceedings, 1993. X, 388 pages. 1993.
- Vol. 754: H. D. Pfeiffer, T. E. Nagle (Eds.), Conceptual Structures: Theory and Implementation. Proceedings, 1992. IX, 327 pages. 1993. (Subseries LNAI).
- Vol. 755: B. Möller, H. Partsch, S. Schuman (Eds.), Formal Program Development. Proceedings. VII, 371 pages. 1993.
- Vol. 756: J. Pieprzyk, B. Sadeghiyan, Design of Hashing Algorithms. XV, 194 pages. 1993.
- Vol. 757: U. Banerjee, D. Gelernter, A. Nicolau, D. Padua (Eds.), Languages and Compilers for Parallel Computing. Proceedings, 1992. X, 576 pages. 1993.
- Vol. 758: M. Teillaud, Towards Dynamic Randomized Algorithms in Computational Geometry. IX, 157 pages. 1993.
- Vol. 759: N. R. Adam, B. K. Bhargava (Eds.), Advanced Database Systems. XV, 451 pages. 1993.
- Vol. 760: S. Ceri, K. Tanaka, S. Tsur (Eds.), Deductive and Object-Oriented Databases. Proceedings, 1993. XII, 488 pages. 1993.
- Vol. 761: R. K. Shyamasundar (Ed.), Foundations of Software Technology and Theoretical Computer Science. Proceedings, 1993. XIV, 456 pages. 1993.
- Vol. 762: K. W. Ng, P. Raghavan, N. V. Balasubramanian, F. Y. L. Chin (Eds.), Algorithms and Computation. Proceedings, 1993. XIII, 542 pages. 1993.
- Vol. 763: F. Pichler, R. Moreno Díaz (Eds.), Computer Aided Systems Theory – EUROCAST '93. Proceedings, 1993. IX, 451 pages. 1994.
- Vol. 764: G. Wagner, Vivid Logic. XII, 148 pages. 1994. (Subseries LNAI).
- Vol. 765: T. Helleseth (Ed.), Advances in Cryptology – EUROCRYPT '93. Proceedings, 1993. X, 467 pages. 1994.
- Vol. 766: P. R. Van Loocke, The Dynamics of Concepts. XI, 340 pages. 1994. (Subseries LNAI).
- Vol. 767: M. Gogolla, An Extended Entity-Relationship Model. X, 136 pages. 1994.
- Vol. 768: U. Banerjee, D. Gelernter, A. Nicolau, D. Padua (Eds.), Languages and Compilers for Parallel Computing. Proceedings, 1993. XI, 655 pages. 1994.
- Vol. 769: J. L. Nazareth, The Newton-Cauchy Framework. XII, 101 pages. 1994.
- Vol. 770: P. Haddawy (Representing Plans Under Uncertainty. X, 129 pages. 1994. (Subseries LNAI).
- Vol. 771: G. Tomas, C. W. Ueberhuber, Visualization of Scientific Parallel Programs. XI, 310 pages. 1994.
- Vol. 772: B. C. Warboys (Ed.), Software Process Technology. Proceedings, 1994. IX, 275 pages. 1994.
- Vol. 773: D. R. Stinson (Ed.), Advances in Cryptology – CRYPTO '93. Proceedings, 1993. X, 492 pages. 1994.
- Vol. 774: M. Banâtre, P. A. Lee (Eds.), Hardware and Software Architectures for Fault Tolerance. XIII, 311 pages. 1994.
- Vol. 775: P. Enjalbert, E. W. Mayr, K. W. Wagner (Eds.), STACS 94. Proceedings, 1994. XIV, 782 pages. 1994.
- Vol. 776: H. J. Schneider, H. Ehrig (Eds.), Graph Transformations in Computer Science. Proceedings, 1993. VIII, 395 pages. 1994.
- Vol. 777: K. von Luck, H. Marburger (Eds.), Management and Processing of Complex Data Structures. Proceedings, 1994. VII, 220 pages. 1994.
- Vol. 778: M. Bonuccelli, P. Crescenzi, R. Petreschi (Eds.), Algorithms and Complexity. Proceedings, 1994. VIII, 222 pages. 1994.
- Vol. 779: M. Jarke, J. Bubenko, K. Jeffery (Eds.), Advances in Database Technology — EDBT '94. Proceedings, 1994. XII, 406 pages. 1994.
- Vol. 780: J. J. Joyce, C.-J. H. Seger (Eds.), Higher Order Logic Theorem Proving and Its Applications. Proceedings, 1993. X, 518 pages. 1994.
- Vol. 781: G. Cohen, S. Litsyn, A. Lobstein, G. Zémor (Eds.), Algebraic Coding. Proceedings, 1993. XII, 326 pages. 1994.
- Vol. 782: J. Gutknecht (Ed.), Programming Languages and System Architectures. Proceedings, 1994. X, 344 pages. 1994.
- Vol. 783: C. G. Günther (Ed.), Mobile Communications. Proceedings, 1994. XVI, 564 pages. 1994.
- Vol. 784: F. Bergadano, L. De Raedt (Eds.), Machine Learning – ECML-94. Proceedings, 1994. XI, 439 pages. 1994. (Subseries LNAI).
- Vol. 785: H. Ehrig, F. Orejas (Eds.), Recent Trends in Data Type Specification. Proceedings, 1992. VIII, 350 pages. 1994.
- Vol. 786: P. A. Fritzson (Ed.), Compiler Construction. Proceedings, 1994. XI, 451 pages. 1994.
- Vol. 787: S. Tison (Ed.), Trees in Algebra and Programming – CAAP '94. Proceedings, 1994. X, 351 pages. 1994.
- Vol. 788: D. Sannella (Ed.), Programming Languages and Systems – ESOP '94. Proceedings, 1994. VIII, 516 pages. 1994.
- Vol. 789: M. Hagiya, J. C. Mitchell (Eds.), Theoretical Aspects of Computer Software. Proceedings, 1994. XI, 887 pages. 1994.