Lecture Notes in Computer Science 1782

Edited by G. Goos, J. Hartmanis and J. van Leeuwen

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

Gert Smolka (Ed.)

Programming Languages and Systems

9th European Symposium on Programming, ESOP 2000
Held as Part of the Joint European Conferences
on Theory and Practice of Software, ETAPS 2000
Berlin, Germany, March 25 – April 2, 2000
Proceedings

Springer

Volume Editor

Gert Smolka
University of Saarland
Programming Systems Lab, Building 45,
P. O. Box 15 11 50, 66041 Saarbrücken, Germany
E-mail: smolka@ps.uni-sb.de

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Programming languages and systems : proceedings / 9th European Symposium on Programming, ESOP 2000, held as part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000 / Gert Smolka (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 2000
(Lecture notes in computer science ; Vol. 1782)
ISBN 3-540-67262-1

CR Subject Classification (1991): D.3, D.1-2, F.3, F.4, E.1

ISSN 0302-9743
ISBN 3-540-67262-1 Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by PTP-Berlin, Stefan Sossna
Printed on acid-free paper SPIN 10719936 06/3142 5 4 3 2 1 0

Foreword

ETAPS 2000 was the third instance of the European Joint Conferences on Theory and Practice of Software. ETAPS is an annual federated conference that was established in 1998 by combining a number of existing and new conferences. This year it comprised five conferences (FOSSACS, FASE, ESOP, CC, TACAS), five satellite workshops (CBS, CMCS, CoFI, GRATRA, INT), seven invited lectures, a panel discussion, and ten tutorials.

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

ETAPS is a loose confederation in which each event retains its own identity, with a separate program committee and independent proceedings. Its format is open-ended, allowing it to grow and evolve as time goes by. Contributed talks and system demonstrations are in synchronized parallel sessions, with invited lectures in plenary sessions. Two of the invited lectures are reserved for "unifying" talks on topics of interest to the whole range of ETAPS attendees. The aim of cramming all this activity into a single one-week meeting is to create a strong magnet for academic and industrial researchers working on topics within its scope, giving them the opportunity to learn about research in related areas, and thereby to foster new and existing links between work in areas that were formerly addressed in separate meetings. The program of ETAPS 2000 included a public business meeting where participants had the opportunity to learn about the present and future organization of ETAPS and to express their opinions about what is bad, what is good, and what might be improved.

ETAPS 2000 was hosted by the Technical University of Berlin and was efficiently organized by the following team:

Bernd Mahr (General Chair)
Hartmut Ehrig (Program Coordination)
Peter Pepper (Organization)
Stefan Jähnichen (Finances)
Radu Popescu-Zeletin (Industrial Relations)

with the assistance of BWO Marketing Service GmbH. The publicity was superbly handled by Doris Fähndrich of the TU Berlin with assistance from the ETAPS publicity chair, Andreas Podelski. Overall planning for ETAPS conferences is the responsibility of the ETAPS steering committee, whose current membership is:

Egidio Astesiano (Genova), Jan Bergstra (Amsterdam), Pierpaolo Degano (Pisa), Hartmut Ehrig (Berlin), José Fiadeiro (Lisbon), Marie-Claude Gaudel (Paris), Susanne Graf (Grenoble), Furio Honsell (Udine), Heinrich Hußmann (Dresden), Stefan Jähnichen (Berlin), Paul Klint (Amsterdam), Tom Maibaum (London), Tiziana Margaria (Dortmund), Ugo Montanari (Pisa), Hanne Riis Nielson (Aarhus), Fernando Orejas (Barcelona), Andreas Podelski (Saarbrücken), David Sands (Göteborg), Don Sannella (Edinburgh), Gert Smolka (Saarbrücken), Bernhard Steffen (Dortmund), Wolfgang Thomas (Aachen), Jerzy Tiuryn (Warsaw), David Watt (Glasgow), Reinhard Wilhelm (Saarbrücken)

ETAPS 2000 received generous sponsorship from:

the Institute for Communication and Software Technology of TU Berlin
the European Association for Programming Languages and Systems
the European Association for Theoretical Computer Science
the European Association for Software Development Science
the "High-Level Scientific Conferences" component of the European
Commission's Fifth Framework Programme

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

January 2000

Donald Sannella
ETAPS Steering Committee chairman

Preface

This volume contains the 27 papers presented at ESOP 2000, the Ninth European Symposium on Programming, which took place in Berlin, March 27–31, 2000. The ESOP series originated in 1986 and addresses the design, specification, and analysis of programming languages and programming systems. Since 1998, ESOP has belonged to the ETAPS confederation.

The call for papers of ESOP 2000 encouraged the following topics: programming paradigms and their integration, including concurrent, functional, logic, and object-oriented; computational calculi and semantics; type systems, program analysis, and concomitant constraint systems; program transformation; programming environments and tools.

The volume starts with a contribution from Martin Odersky, the invited speaker of the conference. The remaining 26 papers were selected by the program committee from 84 submissions (almost twice as many as for ESOP 99). With two exceptions, each submission received at least three reviews, done by the program committee members or their subreferees (names appear below). Once the initial reviews were available, we had two weeks for conflict resolution and paper selection, supported by a database system with Web interfaces.

I would like to express my sincere gratitude to Christian Schulte who took care of the software, handled the submissions, tracked the refereeing process, and finally assembled the proceedings. Then, of course, I am grateful to my fellow program committee members, the many additional referees, and the authors of the submitted papers. Finally, I have to thank Don Sannella, who smoothly organized the program at the ETAPS level and relieved me of many organizational burdens.

January 2000 Gert Smolka

Organization

Program Chair

Gert Smolka	UdS, Saarbrücken, Germany

Program Committee

Gerard Boudol	INRIA, Sophia-Antipolis, France
Sophia Drossopoulou	Imperial College, London, UK
Matthias Felleisen	Rice University, Houston, USA
Michael Franz	UC Irvine, USA
Manuel Hermenegildo	TU Madrid, Spain
Xavier Leroy	INRIA Rocquencourt, France
Alan Mycroft	Cambridge University, UK
Martin Odersky	EPF Lausanne, Switzerland
Andreas Podelski	MPI, Saarbrücken, Germany
Gert Smolka	UdS, Saarbrücken, Germany
Peter Thiemann	Uni Freiburg, Germany
Mads Tofte	Uni Copenhagen, Denmark
Pascal Van Hentenryck	Uni Louvain, Belgium

Additional Referees

Martín Abadi, Roberto Amadio, Zena Matilde Ariola, Andrea Asperti, Uwe Assmann, Isabelle Attali, Gilles Barthe, David Basin, Alexander Bockmayr, Maurice Bruynooghe, Francisco Bueno, Egon Börger, Robert Cartwright, Giuseppe Castagna, Ilaria Castellani, Witold Charatonik, Olaf Chitil, Agostino Cortesi, Patrick Cousot, Roy Crole, Paul Cunningham, Silvano Dal Zilio, Saumya Debray, Bart Demoen, Damien Doligez, Thomas Ehrhard, Jérôme Feret, Gilberto Filé, Ian Foster, Cédric Fournet, Peter H. Froehlich, Martin Fränzle, María García de la Banda, Roberto Giacobazzi, Jens Christian Godskesen, Georges Gonthier, Andy Gordon, Susanne Graf, Niels Hallenberg, Chris Hankin, Nevin Heintze, Simon Helsen, Angel Herranz, Ralf Hinze, Sebastian Hunt, Graham Hutton, Jean-Marie Jacquet, Suresh Jagannathan, C. B. Jay, Neil D. Jones, Antonios Kakas, Sam Kamin, Andy King, Jan Willem Klop, Povl Koch, Shriram Krishnamurthi, Herbert Kuchen, Arun Lakhotia, David Ephraim Larkin, Ziemowit Laski, Baudouin Le Charlier, Fabrice Le Fessant, K. Rustan M. Leino, Jean-Jacques Lévy, Michael Maher, Jan Maluszynski, John Maraist, Luc Maranget, Mircea Marin, Julio Mariño, Kim Marriott, Laurent Mauborgne, Erik Meijer, Massimo Merro, Laurent Michel, Yasuhiko Minamide, Eugenio Moggi, Andrew Moran, Juan José Moreno Navarro, Anders Møller, Peter Møller

Neergaard, Lee Naish, Uwe Nestmann, Flemming Nielson, Jukka Paakki, Jens Palsberg, Ross Paterson, Alberto Pettorossi, Iain Phillips, Enrico Pontelli, François Pottier, Germàn Puebla, Christian Queinnec, Laurent Regnier, John Reppy, Hanne Riis Nielson, David Rosenblueth, Andreas Rossberg, Abhik Roychowdhury, Albert Rubio, Radu Rugina, Claudio Vittorio Russo, Didier Rémy, Michel Rüher, Amr Sabry, Beverly Sanders, Davide Sangiorgi, Hiroyuki Sato, David Schmidt, Wolfgang Schreiner, Christian Schulte, Peter Sestoft, Zhong Shao, Richard Sharp, Yu Shi, Harald Sondergaard, Fausto Spoto, Harini Srinivasan, Paul Steckler, Peter Stuckey, Jean-Ferdinand Susini, Don Syme, Sophie Tison, Jan Vitek, Philip Wadler, David S. Warren, Reinhard Wilhelm, Burkhard Wolff, Andrew Wright, Christoph Zenger, Matthias Zenger, Elena Zucca, Frank S. de Boer.

Table of Contents

Invited Paper

Regular Papers

Functional Nets

Martin Odersky

École Polytechnique Fédérale de Lausanne

Abstract. Functional nets combine key ideas of functional programming and Petri nets to yield a simple and general programming notation. They have their theoretical foundation in Join calculus. This paper presents functional nets, reviews Join calculus, and shows how the two relate.

1 Introduction

Functional nets are a way to think about programs and computation which is born from a fusion of the essential ideas of functional programming and Petri nets. As in functional programming, the basic computation step in a functional net rewrites function applications to function bodies. As in Petri-Nets, a rewrite step can require the combined presence of several inputs (where in this case inputs are function applications). This fusion of ideas from two different areas results in a style of programming which is at the same time very simple and very expressive.

Functional nets have a theoretical foundation in *join calculus* [15,16]. They have the same relation to join calculus as classical functional programming has to λ-calculus. That is, functional nets constitute a programming method which derives much of its simplicity and elegance from close connections to a fundamental underlying calculus. λ-calculus [10,5] is ideally suited as a basis for functional programs, but it can support mutable state only indirectly, and nondeterminism and concurrency not at all. The pair of join calculus and functional nets has much broader applicability – functional, imperative and concurrent program constructions are supported with equal ease.

The purpose of this paper is two-fold. First, it aims to promote functional nets as an interesting programming method of wide applicability. We present a sequence of examples which show how functional nets can concisely model key constructs of functional, imperative, and concurrent programming, and how they often lead to better solutions to programming problems than conventional methods.

Second, the paper develops concepts to link our programming notation of functional nets with the underlying calculus. To scale up from a calculus to a programming language, it is essential to have a means of aggregating functions and data. We introduce *qualified definitions* as a new syntactic construct for aggregation. In the context of functional nets, qualified definitions provide more flexible control over visibility and initialization than the more conventional

G. Smolka (Ed.): ESOP/ETAPS 2000, LNCS 1782, pp. 1–25, 2000.

record- or object-constructors. They are also an excellent fit to the underlying join calculus, since they maintain the convention that every value has a name. We will present object-based join calculus, an extension of join calculus with qualified definitions. This extension comes at surprisingly low cost, in the sense that the calculus needs to be changed only minimally and all concepts carry over unchanged. By contrast, conventional record constructors would create anonymous values, which would be at odds with the name-passing nature of join.

The notation for writing examples of functional nets is derived from Silk, a small language which maps directly into our object-based extension of join. An implementation of Silk is publicly available. There are also other languages which are based in some form on join calculus, and which express the constructs of functional nets in a different way, e.g. Join[17] or JoCaml[14]. We have chosen to develop and present a new notation since we wanted to support both functions and objects in a way which was as simple as possible.

As every program notation should be, functional nets are intended to be strongly typed, in the sense that all type errors should be detected rather than leading to unspecified behavior. We leave open whether type checking is done statically at compile time or dynamically at run time. Our examples do not mention types, but they are all of a form that would be checkable using a standard type system with recursive records, subtyping and polymorphism.

The rest of this paper is structured as follows. Section 2 introduces functional nets and qualified definitions. Sections 3 and 4 show how common functional and imperative programming patterns can be modeled as functional nets. Section 5 discusses concurrency and shows how functional nets model a wide spectrum of process synchronization techniques. Section 6 introduces object-based join calculus as the formal foundation of functional nets. Section 7 discusses how the programming notation used in previous sections can be encoded in this calculus. Section 8 discusses related work and concludes.

2 A First Example

Consider the task of implementing a one-place buffer, which connects producers and consumers of data. Producers call a function put to deposit data into the buffer while consumers call a function get to retrieve data from the buffer. There can be at most one datum in the buffer at any one time. A put operation on a buffer which is already full blocks until the buffer is empty. Likewise, a get on an empty buffer blocks until the buffer is full. This specification is realized by the following simple functional net:

```
def get & full x    = x & empty,
    put x & empty = () & full x
```

The net contains two definitions which together define four functions. Two of the functions, put and get, are meant to be called from the producer and consumer clients of the buffer. The other two, full and empty, reflect the buffer's internal state, and should be called only from within the buffer.

Function put takes a single argument, x. We often write a function argument without surrounding parentheses, e.g. put x instead of put(x). We also admit functions like get that do not take any argument; one can imagine that every occurrence of such a function is augmented by an implicit empty tuple as argument, e.g. get becomes get().

The two equations define *rewrite rules.* A set of function calls that matches the left-hand side of an equation may be rewritten to the equation's right-hand side. The & symbol denotes parallel composition. We sometimes call & a *fork* if it appears on an equation's right-hand side, and a *join* if it appears on the left. Consequently, the left-hand sides of equations are also called *join patterns.*

For instance, the equation

```
get & full x  =  x & empty
```

states that if there are two concurrent calls, one to get and the other to full x for some value x, then those calls may be rewritten to the expression x & empty. That expression returns x as get's result and in parallel calls function empty. Unlike get, empty does not return a result; it's sole purpose is to enable via the second rewrite rule calls to put to proceed. We call result-returning functions like get *synchronous*, whereas functions like empty are called *asynchronous.*

In general, only the leftmost operand of a fork or a join can return a result. All function symbols of a left-hand side but the first one are asynchronous. Likewise, all operands of a fork except the first one are asynchronous or their result is discarded.

It's now easy to interpret the second rewrite rule,

```
put x & empty  =  () & full x
```

This rule states that two concurrent calls to put x & empty and may be rewritten to () & full x. The result part of that expression is the unit value (); it signals termination and otherwise carries no interesting information.

Clients of the buffer still need to initialize it by calling empty. A simple usage of the one-place buffer is illustrated in the following example.

```
def get & full x     =  x  & empty,
    put x & empty =  () & full x;

put 1 &
  ( val y = get ; val r = y + y ; print r ; put r ) &
  ( val z = get ; val r = y * y ; print r; put r ) &
  empty
```

Besides the initializer empty there are three client processes composed in parallel. One process puts the number 1 into the buffer. The other two processes both try to get the buffer's contents and put back a modified value. The construct

```
val y = get ; ...
```

evaluates the right-hand side expression get and defines y as a name for the resulting value. The defined name y remains visible in the expression following the semicolon. By contrast, if we had written **def** y = get; ... we would have defined a function y, which each time it was called would call in turn get. The definition itself would not evaluate anything.

As usual, a semicolon between expressions stands for sequencing. The combined expression print r; put r first prints its argument r and then puts it into the buffer.

The sequence in which the client processes in the above example execute is arbitrary, controlled only by the buffer's rewrite rules. The effect of running the example program is hence the output of two numbers, either (2, 4) or (1, 2), depending which client process came first.

Objects The previous example mixed the definition of a one-place buffer and the client program using it. A better de-coupling is obtained by defining a constructor function for one-place buffers. The constructor, together with a program using it can be written as follows.

```
def newBuffer = {
  def get & full x    =  x  & empty,
      put x & empty =  () & full x;
  (get, put) & empty
};
val (get', put') = newBuffer;
put' 1 &
  ( val y = get' ; val r = y + y ; print r ; put' r ) &
  ( val z = get' ; val r = y * y ; print r ; put' r )
```

The defining equations of a one-place buffer are now local to a block, from which the pair of locally defined functions get and put is returned. Parallel to returning the result the buffer is initialized by calling empty. The initializer empty is now part of the constructor function; clients no longer can call it explicitly, since empty is defined in a local block and not returned as result of that block. Hence, newBuffer defines an object with externally visible methods get and put and private methods empty and full. The object is represented by a tuple which contains all externally visible methods.

This representation is feasible as long as objects have only a few externally visible methods, but for objects with many methods the resulting long tuples quickly become unmanageable. Furthermore, tuples do not support a notion of subtyping, where an object can be substituted for another one with fewer methods. We therefore introduce *records* as a more suitable means of aggregation where individual methods can be accessed by their names, and subtyping is possible.

The idiom for record access is standard. If r denotes a record, then $r.f$ denotes the field of r named f. We also call references of the form $r.f$ *qualified names*. The idiom for record creation is less conventional. In most programming languages, records are defined by enumerating all field names with their values. This notion interacts poorly with the forms of definitions employed in functional nets. In a functional net, one often wants to export only some of the functions defined in a join pattern whereas other functions should remain hidden. Moreover, it is often necessary to call some of the hidden functions as part of the object's initialization.

To streamline the construction of objects, we introduce qualified names not only for record accesses, but also for record definitions. For instance, here is a re-formulation of the newBuffer function using qualified definitions.

```
def newBuffer = {
   def this.get & full x       =  x  & empty,
        this.put x & empty  =  () & full x;
   this & empty
};
val buf = newBuffer;
buf.put 1 &
  ( val y = buf.get ; val r = y + y ; print r ; buf.put r ) &
  ( val z = buf.get ; val r = y * y ; print r ; buf.put r )
```

Note the occurrence of the qualified names this.get and this.put on the left-hand side of the local definitions. These definitions introduce three local names:

– the local name this, which denotes a record with two fields, get and put, and
– local names empty and full, which denote functions.

Note that the naming of this is arbitrary, any other name would work equally well. Note also that empty and full are not part of the record returned from newRef, so that they can be accessed only internally.

The identifiers which occur before a period in a join pattern always define new record names, which are defined only in the enclosing definition. It is not possible to use this form of qualified definition to add new fields to a record defined elsewhere.

Some Notes on Syntax We assume the following order of precedence, from strong to weak:

```
( ) and (.)  ,  (&)  ,  (=)  ,  (,)  ,  (;)  .
```

That is, function application and selection bind strongest, followed by parallel composition, followed by the equal sign, followed by comma, and finally followed by semicolon. Function application and selection are left associative, & is associative, and ; is right associatve. Other standard operators such as +, $*$, == fall between function application and & in their usual order of precedence. When precedence risks being unclear, we'll use parentheses to disambiguate.

As a syntactic convenience, we allow indentation instead of ;-separators inside blocks delimited with braces { and }. Except for the significance of indentation, braces are equivalent to parentheses. The rules are as follows: (1) in a block delimited with braces, a semicolon is inserted in front of any non-empty line which starts at the same indentation level as the first symbol following the opening brace, provided the symbol after the insertion point can start an expression or definition. The only modification to this rule is: (2) if inserted semicolons would separate two **def** blocks, yielding **def** D_1 ; **def** D_2 say, then the two **def** blocks are instead merged into a single block, i.e. **def** D_1, D_2. (3) The top level program is treated like a block delimited with braces, i.e. indentation is significant.

With these rules, the newBuffer example can alternatively be written as follows.

```
def newBuffer = {
    def this.get & full x       =  x  & empty
    def this.put x & empty  =  () & full x
    this & empty
}
val buf = newBuffer
buf.put 1 &
  { val y = buf.get ; val r = y + y ; print r ; buf.put r } &
  { val z = buf.get ; val r = y ∗ y ; print r ; buf.put r }
```

A common special case of a qualified definition is the definition of a record with only externally visible methods:

```
( def this.f = ... , this.g = ... ; this )
```

This idiom can be abbreviated by omitting the this qualifier and writing only the definitions.

```
( def f = ... , g = ... )
```

3 Functional Programming

A functional net that does not contain any occurrences of & is a purely functional program. For example, here's the factorial function written as a functional net.

```
def factorial n  =  if (n == 0) 1
                    else n ∗ factorial (n–1)
```

Except for minor syntactical details, there's nothing which distinguishes this program from a program written in a functional language like Haskell or ML. We assume that evaluation of function arguments is strict: In the call f (g x), g x will be evaluated first and its value will be passed to f.

Functional programs often work with recursive data structures such as trees and lists. In Lisp or Scheme such data structures are primitive S-expressions, whereas in ML or Haskell they are definable as algebraic data types. Our functional net notation does not have a primitive tree type, nor has it constructs for defining algebraic data types and for pattern matching their values. It does not need to, since these constructs can be represented with records, using the *Visitor* pattern[18].

The visitor pattern is the object-oriented version of the standard Church encoding of algebraic data types. A visitor encodes the branches of a pattern matching case expression. It is represented as a record with one method for each branch. For instance, a visitor for lists would always have two methods:

```
def Nil = ...
def Cons (x, xs) = ...
```

The intention is that our translation of pattern matching would call either the Nil method or the Cons method of a given visitor, depending what kind of list was encountered. If the encountered list resulted from a Cons we also need to pass the arguments of the original Cons to the visitor's Cons.

Assume we have already defined a method match for lists that takes a list visitor as argument and has the behavior just described. Then one could write an isEmpty test function over lists as follows:

```
def isEmpty xs = xs.match {
    def Nil = true
    def Cons (x, xs1) = false
}
```

More generally, every function over lists can be defined in terms of match. So, in order to define a record which represents a list, all we need to do is to provide a match method. How should match be defined? Clearly, its behavior will depend on whether it is called on an empty or non-empty list. Therefore, we define two list constructors Nil and Cons, with two different different implementations for match. The implementations are straightforward:

```
val List = {
    def Nil          =  { def match v = v.Nil }
    def Cons (x, xs) =  { def match v = v.Cons (x, xs) }
}
```

In each case, match simply calls the appropriate method of its visitor argument v, passing any parameters along. We have chosen to wrap the Nil and Cons constructors in another record, named List. List acts as a module, which provides the constructors of the list data type. Clients of the List module then construct lists using qualified names List.Nil and List.Cons. Example:

```
def concat (xs, ys) = xs.match {
    def Nil = ys
    def Cons (x, xs) = List.Cons (x, concat (xs1, ys))
}
```

Note that the qualification with List lets us distinguish the constructor Cons, defined in List, from the visitor method Cons, which is defined locally.

4 Imperative Programming

Imperative programming extends purely functional programming with the addition of mutable variables. A mutable variable can be modeled as a reference cell object, which can be constructed as follows.

```
def newRef initial  =  {
   def this.value      & state x   =  x  & state x,
       this.update y  & state x   =  () & state y

   this & state initial
}
```

The structure of these definitions is similar to the one-place buffer in Section 2. The two synchronous functions value and update access and update the variable's current value. The asynchronous function state serves to remember the variable's current value. The reference cell is initialized by calling state with the initial value.

Here is a simple example of how references are used:

```
val count = newRef 0
def increment  =  count.update (count.value + 1)
increment
```

Building on reference cell objects, we can express the usual variable access notation of imperative languages by two simple syntactic expansions:

```
var x := E      expands to      val _x = newRef E ; def x = _x.value
x := E          expands to      _x.update E
```

The count example above could then be written more conventionally as follows.

```
var count := 0
def increment  =  count := count + 1
```

In the object-oriented design and programming area, an object is often characterized as having "state, behavior, and identity". Our encoding of objects expresses state as a collection of applications of private asynchronous functions, and behavior as a collection of externally visible functions. But what about identity? If functional net objects had an observable identity it should be possible to define a method eq which returns true if and only if its argument is the same object as the current object. Here "sameness" has to be interpreted as "created by the same operation", structural equality is not enough. E.g., assuming that the – as yet hypothetical – eq method was added to reference objects, it should be possible to write **val** (r1, r2) = (newRef 0, newRef 0) and to have r1.eq(r1) == true and r1.eq(r2) == false.

Functional nets have no predefined operation which tests whether two names or references are the same. However, it is still possible to implement an eq method. Here's our first attempt, which still needs to be refined later.

$$f_1 \ \& \ \ldots \ \& \ f_m \quad = \quad g_1 \ \& \ \ldots \ \& \ g_n$$

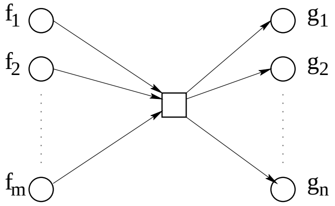


Fig. 1. Analogy to Petri nets

```
def newObjectWithIdentity  =  {
   def this.eq other      & flag x  =  resetFlag (other.testFlag & flag true)
       this.testFlag      & flag x  =  x & flag x
       resetFlag result   & flag x  =  x & flag false
   this & flag false
}
```

This defines a generator function for objects with an eq method that tests for identity. The implementation of eq relies on three helper functions, flag, testFlag, and resetFlag. Between calls to the eq method, flag false is always asserted. The trick is that the eq method asserts flag true and at the same time tests whether other.flag is true. If the current object and the other object are the same, that test will yield true. On the other hand, if the current object and the other object are different, the test will yield false, provided there is not at the same time another ongoing eq operation on object other. Hence, we have arrived at a solution of our problem, provided we can prevent overlapping eq operations on the same objects. In the next section, we will develop techniques to do so.

5 Concurrency

The previous sections have shown how functional nets can express sequential programs, both in functional and in imperative style. In this section, we will show their utility in expressing common patterns of concurrent program execution.

Functional nets support an resource-based view of concurrency, where calls model resources, & expresses conjunction of resources, and a definition acts as a rewrite rule which maps input sets of resources into output sets of resources. This view is very similar to the one of Petri nets [29,32]. In fact, there are direct analogies between the elements of Petri nets and functional nets. This is illustrated in Figure 1.

A *transition* in a Petri net corresponds to a rewrite rule in a functional net. A *place* in a Petri net corresponds to a function symbol applied to some (formal or actual) arguments. A *token* in a Petri net corresponds to some actual call during the execution of a functional net (in analogy to Petri nets, we will also call applications of asynchronous functions *tokens*). The basic execution step of a Petri net is the firing of a transition which has as a precondition that all in-going places have tokens in them. Quite similarly, the basic execution step of a functional net is a rewriting according to some rewrite rule, which has as a precondition that all function symbols of the rule's left-hand side have matching calls.

Functional nets are considerably more powerful than conventional Petri nets, however. First, function applications in a functional net can have arguments, whereas tokens in a Petri net are unstructured. Second, functions in a functional net can be higher-order, in that they can have functions as their arguments. In Petri nets, such self-referentiality is not possible. Third, definitions in a functional net can be nested inside rewrite rules, such that evolving net topologies are possible. A Petri-net, on the other hand, has a fixed connection structure.

Colored Petri nets [24] let one pass parameters along the arrows connecting places with transitions. These nets are equivalent to first-order functional nets with only global definitions. They still cannot express the higher-order and evolution aspects of functional nets. Bussi and Asperti have translated join calculus ideas into standard Petri net formalisms. Their mobile Petri nets [4] support first-class functions and evolution, and drop at the same time the locality restrictions of join calculus and functional nets. That is, their notation separates function name introduction from rewrite rule definition, and allows a function to be defined collectively by several unrelated definitions.

In the following, we will present several well-known schemes for process synchronization and how they each can be expressed as functional nets.

Semaphores A common mechanism for process synchronization is a *lock* (or: *semaphore*). A lock offers two atomic actions: getLock and releaseLock. Here's the implementation of a lock as a functional net:

```
def newLock = {
    def this.getLock & this.releaseLock  =  ()
    this & this.releaseLock
}
```

A typical usage of a semaphore would be:

```
val s = newLock ; ...
s.getLock ; "< critical region >" ; s.releaseLock
```

With semaphores, we can now complete our example to define objects with identity:

```
val global = newLock
def newObjectWithIdentity  =  {
    def this.eq other = global.getLock ; this.testEq other ; global.releaseLock
        this.testEq other & flag x  =  resetFlag (other.testFlag & flag true)
        this.testFlag       & flag x  =  x & flag x
        resetFlag result   & flag x  =  x & flag false

    this & flag false
}
```

This code makes use of a global lock to serialize all calls of eq methods. This is admittedly a brute force approach to mutual exclusion, which also serializes calls to eq over disjoint pairs of objects. A more refined locking strategy is hard to come by, however. Conceptually, a critical region consists of a pair of objects which both have to be locked. A naive approach would lock first one object, then the other. But this would carry the risk of deadlocks, when two concurrent eq operations involve the same objects, but in different order.

Asynchronous Channels Quite similar to a semaphore is the definition of an asynchronous channel with two operations, read and write:

```
def newAsyncChannel = {
   def this.read & this.write x  =  x
   this
}
```

Asynchronous channels are the fundamental communication primitive of asynchronous π calculus [8,23] and languages based on it, e.g. Pict[30] or Piccola[1]. A typical usage scenario of an asynchronous channel would be:

```
val c = newAsyncChannel
def producer = {
   var x := 1
   while (true) { val y := x ; x := x + 1 & c.write y }
}
def consumer = {
   while (true) { val y = c.read ; print y }
}
producer & consumer
```

The producer in the above scenario writes consecutive integers to the channel c which are read and printed by the consumer. The writing is done asynchronously, in parallel to the rest of the body of the producer's while loop. Hence, there is no guarantee that numbers will be read and printed in the same order as they were written.

Synchronous Channels A potential problem with the previous example is that the producer might produce data much more rapidly than the consumer consumes them. In this case, the number of pending write operations might increase indefinitely, or until memory was exhausted. The problem can be avoided by connecting producer and consumer with a synchronous channel.

In a synchronous channel, both reads and writes return and each operation blocks until the other operation is called. Synchronous channels are the fundamental communication primitive of classical π-calculus[27]. They can be represented as functional nets as follows.

```
def newSyncChannel = {
  def this.read & noReads      =  read1 & read2,
      this.write x & noWrites  =  write1 & write2 x,
      read1 & write2 x         =  x & noWrites,
      write1 & read2           =  () & noReads
  this
}
```

This implementation is more involved than the one for asynchronous channels. The added complexity stems from the fact that a synchronous channel connects two synchronous operations, yet in each join pattern there can be only one function that returns. Our solution is similar to a double handshake protocol. It splits up read and write into two sub-operations each, read1, read2 and write1, write2. The sub-operations are then matched in two join patterns, in opposite senses. In one pattern it is the read sub-operation which returns whereas in the second one it is the write sub-operation. The noReads and noWrites tokens are necessary for serializing reads and writes, so that a second write operation can only start after the previous read operation is finished and vice versa. With synchronous channels, our producer/consumer example can be written as follows.

```
val c = newSyncChannel
def producer = {
  var x := 1
  while (true) { c.write x  ;  x := x + 1 }
}
def consumer = {
  while (true) { val y = c.read  ;  print y }
}
producer & consumer
```

Monitors Another scheme for process communication is to use a common store made up of mutable variables, and to use mutual exclusion mechanisms to prevent multiple processes from updating the same variable at the same time. A simple mutual exclusion mechanism is the *monitor* [20,21] which ensures that only one of a set of functions f_1, ..., f_k can be active at any one time. A monitor is easily represented using an additional asynchronous function, turn. The turn token acts as a resource which is consumed at the start of each function f_i and which is reproduced at the end:

```
def f_1 & turn  =   ... ; turn,
                     ⋮
    f_k & turn  =   ... ; turn
```

For instance, here is an example of a counter which can be incremented and decremented:

```
def newBiCounter = {
    var count := 0
    def this.increment & turn  =  count := count + 1 ; turn
    def this.decrement & turn  =  count := count − 1 ; turn
    this
}
```

Readers and Writers A more complex form of synchronization distinguishes between *readers* which access a common resource without modifying it and *writers* which can both access and modify it. To synchronize readers and writers we need to implement operations startRead, startWrite, endRead, endWrite, such that:

- there can be multiple concurrent readers,
- there can only be one writer at one time,
- pending write requests have priority over pending read requests, but don't preempt ongoing read operations.

This form of access control is common in databases. It can be implemented using traditional synchronization mechanisms such as semaphores, but this is far from trivial. We arrive at a functional net solution to the problem in two steps.

The initial solution is given at the top of Figure 2. We make use of two auxiliary tokens. The token readers n keeps track in its argument n of the number of *ongoing* reads, while writers n keeps track in n of the number of *pending* writes. A startRead operation requires that there are no pending writes to proceed, i.e. writers 0 must be asserted. In that case, startRead continues with startRead1, which reasserts writers 0, increments the number of ongoing readers, and returns to its caller. By contrast, a startWrite operation immediately increments the number of pending writes. It then continues with startWrite1, which waits for the number of readers to be 0 and then returns. Note the almost-symmetry between startRead and startWrite, where the different order of actions reflects the different priorities of readers and writers.

This solution is simple enough to trust its correctness. But the present formulation is not yet valid Silk because we have made use of numeric arguments in join patterns. For instance readers 0 expresses the condition that the number of readers is zero. We arrive at an equivalent formulation in Silk through factorization. A function such as readers which represents a condition is split into several sub-functions which together partition the condition into its cases of interest. In our case we should have a token noReaders which expresses the fact that there are no ongoing reads as well as a token readers n, where n is now required to be positive. Similarly, writers n is now augmented by a case noWriters. After splitting and introducing the necessary case distinctions, one obtains the functional net listed at the bottom of Figure 2.

Initial solution:

```
    def this.startRead   & writers 0  =  startRead1,
             startRead1  & readers n  =  () & writers 0 & readers (n+1),
         this.startWrite & writers n  =  startWrite1 & writers (n+1),
             startWrite1 & readers 0  =  (),

         this.endRead    & readers n  =  readers (n–1),
         this.endWrite   & writers n  =  writers (n–1) & readers 0

    this & readers 0 & writers 0
  }
```

After factorization:

```
  def newReadersWriters = {
    def this.startRead   & noWriters  =  startRead1,
             startRead1  & noReaders  =  () & noWriters & readers 1,
             startRead1  & readers n  =  () & noWriters & readers (n+1),

         this.startWrite & noWriters  =  startWrite1 & writers 1,
         this.startWrite & writers n  =  startWrite1 & writers (n+1),
             startWrite1 & noReaders  =  (),

         this.endRead    & readers n  =  if (n == 1) noReaders
                                         else readers (n–1),
         this.endWrite   & writers n  =  noReaders &
                                         if (n == 1) noWriters
                                         else writers (n–1)

    this & noReaders & noWriters
  }
```

Fig. 2. Readers/writers synchronization

6 Foundations: The Join Calculus

Functional nets have their formal basis in join calculus [15]. We now present this basis, in three stages. In the first stage, we study a subset of join calculcus which can be taken as the formal basis of purely functional programs. This calculus is equivalent to (call-by-value) λ-calculus[31], but takes the opposite position on naming functions. Where λ-calculus knows only anonymous functions, functional join calculus insists that every function have a name. Furthermore, it also insists that every intermediate result be named. As such it is quite similar to common forms of intermediate code found in compilers for functional languages.

The second stage adds fork and join operators to the constructs introduced in the first stage. The calculus developed at this stage is equivalent in principle to the original join calculus, but some syntactical details have changed.

The third stage adds qualified names in definitions and accesses. The calculus developed in this stage can model the object-based functional nets we have used.

Syntax:

Names	$a, b, c, \ldots, x, y, z$		
Terms	M, N	$=$	$\mathbf{def}\, D \,;\, M \;\mid\; x(\tilde{y})$
Definitions	D	$=$	$L = M$
Left-hand sides	L	$=$	$x(\tilde{y})$
Reduction contexts	R	$=$	$[\,] \mid \mathbf{def}\, D \,;\, R$

Structural Equivalence: α-renaming.

Reduction:

$$\mathbf{def}\, x(\tilde{y}) = M \;;\; R[x(\tilde{z})] \quad \rightarrow \quad \mathbf{def}\, x(\tilde{y}) = M \;;\; R[[\tilde{z}/\tilde{y}]M]$$

Fig. 3. Pure functional calculus

All three stages represent functional nets as a reduction system. There is in each case only a single rewrite rule, which is similar to the β-reduction rule of λ-calculus, thus closely matching intuitions of functional programming. By contrast, the original treatment of join calculus is based on a chemical abstract machine[6], a concept well established in concurrency theory. The two versions of join calculus complement each other and are (modulo some minor syntactical details) equivalent.

6.1 Pure Functional Calculus

Figure 3 presents the subset of join calculus which can express purely functional programs. The syntax of this calculus is quite small. A *term* M is either a function application $x(\tilde{y})$ or a term with a local definition, **def** D ; M (we let $\tilde{x}$ stand for a sequence $x_1, \ldots, x_n$ of names, where $n \geq 0$). A *definition* D is a single rewrite rule the form $L = M$. The *left-hand side* L of a rewrite rule is again a function application $x(\tilde{y})$. We require that the formal parameters y_i of a left-hand side are pairwise disjoint. The right-hand side of a rewrite rule is an arbitrary term.

The set of *defined names* $\mathrm{dn}(D)$ of a definition D of the form $x(\tilde{y}) = M$ consists of just the function name x. Its *local names* $\mathrm{ln}(D)$ are the formal parameters $\tilde{y}$. The *free names* $\mathrm{fn}(M)$ of a term M are all names which are not defined by or local to a definition in M. The free names of a definition D are its defined names and any names which are free in the definition's right hand side, yet different from the local names of D. All names occurring in a term M that are not free in M are called *bound* in M. Figure 6 presents a formal definition of these sets for the object-based extension of join calculus.

To avoid unwanted name capture, where free names become bound inadvertently, we will always write terms subject to the following *hygiene condition*: We assume that the set of free and bound names of every term we write are disjoint.

This can always be achieved by a suitable renaming of bound variables, according to the *α-renaming* law. This law lets us rename local and defined names of definitions, provided that the new names do not clash with names which already exist in their scope. It is formalized by two equations. First,

$$\mathbf{def}\, x(\tilde{y}) = M \;;\; N \equiv \mathbf{def}\, u(\tilde{y}) = [u/x]M \;;\; [u/x]N$$

if $u \notin \mathrm{fn}(M) \cup \mathrm{fn}(N)$. Second,

$$\mathbf{def}\, x(\tilde{y}) = M \;;\; N \equiv \mathbf{def}\, x(\tilde{v}) = [\tilde{v}/\tilde{y}]M \;;\; N$$

if $\{\tilde{v}\} \cap \mathrm{fn}(M) = \emptyset$ and the elements of $\tilde{v}$ are pairwise disjoint. Here, $[u/x]$ and $[\tilde{v}/\tilde{y}]$ are substitutions which map x and $\tilde{y}$ to u and $\tilde{v}$. Generally, substitutions are idempotent functions over names which map all but a finite number of names to themselves. The *domain* $\mathrm{dom}(\sigma)$ of a substitution σ is the set of names not mapped to themselves under σ.

Generally, we will give in each case a structural equivalence relation $\equiv$ which is assumed to be reflexive, transitive, and compatible (i.e. closed under formation of contexts). Terms that are related by $\equiv$ are identified with each other. For the purely functional calculus, $\equiv$ is just α-renaming. Extended calculi will have richer notions of structural equivalence.

Execution of terms in our calculus is defined by rewriting. Figure 3 defines a single rewrite rule, which is analogous to β-reduction in λ-calculus. The rule can be sketched as follows:

$$\mathbf{def}\, x(\tilde{y}) = M \;;\; \ldots x(\tilde{z}) \ldots \quad \rightarrow \quad \mathbf{def}\, x(\tilde{y}) = M \;;\; \ldots [\tilde{z}/\tilde{y}]M \ldots$$

That is, if there is an application $x(\tilde{z})$ which matches a definition of x, say $x(\tilde{y}) = M$, then we can rewrite the application to the definition's right hand side M, after replacing formal parameters $\tilde{y}$ by actual arguments $\tilde{z}$.

The above formulation is not yet completely precise because we still have to specify where exactly a reducible application can be located in a term. Clearly, the application must be within the definition's scope. Also, we want to reduce only those applications which are not themselves contained in another local definition. For instance, in

```
def f (x, k) = k x ;
def g (x, k) = f (1, k) ;
f(2, k)
```

we want to reduce only the second application of f, not the first one which is contained in the body of function g. This restriction in the choice of reducible applications avoids potentially unnecessary work. For instance in the code fragment above g is never called, so it would make no sense to reduce its body. More importantly, once we add side-effects to our language, it is essential that the body of a function is executed (i.e. reduced) only when the function is applied.

The characterization of reducible applications can be formalized using the idea of a *reduction context*[1]. A *context* C is a term with a hole, which is written $[\,]$. The expression $C[M]$ denotes the term resulting from filling the hole of the context C with M. A *reduction context* R is a context of a special form, in which the hole can be only at places where a function application would be reducible. The set of possible reduction contexts for our calculus is generated by a simple context free grammar, given in Figure 3. This grammar says that reduction can only take place at the top of a term, or in the scope of some local definitions.

Reduction contexts are used in the formulation of the reduction law in Figure 3. Generally, we let the reduction relation → between terms be the smallest compatible relation that contains the reduction law.

An alternative formulation of the reduction rule abstracts from the concrete substitution operator:

$$\mathbf{def}\, L = M \;;\; R[\sigma L] \quad \rightarrow \quad \mathbf{def}\, L = M \;;\; R[\sigma M]$$

if σ is a substitution from names to names with $\mathrm{dom}(\sigma) \subseteq \mathrm{ln}(L)$.

The advantage of the alternative formulation is that it generalizes readily to the more complex join patterns which will be introduced in the next sub-section.

As an example of functional reduction, consider the following forwarding function:

$$\mathbf{def}\, \mathsf{f(x)} = \mathsf{g(x)} \;;\; \mathsf{f(y)} \quad \rightarrow \quad \mathbf{def}\, \mathsf{f(x)} = \mathsf{g(x)} \;;\; \mathsf{g(y)}$$

A slightly more complex example is the following reduction of a call to an evaluation function, which takes two arguments and applies one to the other:

$$\mathbf{def}\, \mathsf{apply(f,x)} = \mathsf{f(x)} \;;\; \mathsf{apply(print,\ 1)} \quad \rightarrow \quad \mathbf{def}\, \mathsf{apply(f,x)} = \mathsf{f(x)} \;;\; \mathsf{print(1)}$$

6.2 Canonical Join Calculus

Figure 4 presents the standard version of join calculus. Compared to the purely functional subset, there are three syntax additions: First and second, & is now introduced as *fork* operator on terms and as a *join* operator on left-hand sides. Third, definitions can now consist of more than one rewrite rule, so that multiple definitions of the same function symbol are possible.

The latter addition is essentially for convenience, as one can translate every program with definitions consisting of multiple rewrite rules to a program that uses just one rewrite rule for each definition [15]. The convenience is great enough to warrant a syntax extension because the encoding is rather heavy.

The notion of structural equivalence is now more refined than in the purely functional subset. Besides α-renaming, there are three other sets of laws which identify terms. First, the fork operator is assumed to be associative and commutative. Second, the comma operator which conjoins rewrite rules is also taken to

[1] The concept is usually known as under the name "evaluation context" [11], but there's nothing to evaluate here.

Syntax:

Names	$a, b, c, \ldots, x, y, z$		
Terms	M, N	$=$	$\mathbf{def}\, D \;;\, M \;\mid\; x(\tilde{y}) \;\mid\; M \;\&\; N$
Definitions	D	$=$	$L = M \;\mid\; D, D \;\mid\; \epsilon$
Left-hand sides	L	$=$	$x(\tilde{y}) \;\mid\; L \;\&\; L$
Reduction contexts	R	$=$	$[\,] \;\mid\; \mathbf{def}\, D \;;\, R \;\mid\; R \;\&\; M \;\mid\; M \;\&\; R$

Structural Equivalence: α-renaming +

1. $(\&)$ on terms is associative and commutative:
$$M_1 \;\&\; M_2 \equiv M_2 \;\&\; M_1$$
$$M_1 \;\&\; (M_2 \;\&\; M_3) \equiv (M_1 \;\&\; M_2) \;\&\; M_3$$
2. $(,)$ on definitions is associative and commutative with ϵ as an identity:
$$D_1, D_2 \equiv D_2, D_1$$
$$D_1, (D_2, D_3) \equiv (D_1, D_2), D_3$$
$$D, \epsilon \equiv D$$
3. Scope extrusion:
$$(\mathbf{def}\, D \;;\, M) \;\&\; N \equiv \mathbf{def}\, D \;;\, (M \;\&\; N) \qquad \text{if } \mathrm{dn}(D) \cap \mathrm{fn}(N) = \emptyset.$$

Reduction:

$$\mathbf{def}\, D, L = M \;;\, R[\sigma L] \quad \rightarrow \quad \mathbf{def}\, D, L = M \;;\, R[\sigma M]$$

where σ is a substitution from names to names with $\mathrm{dom}(\sigma) \subseteq \mathrm{ln}(L)$.

Fig. 4. Canonical join calculus

be associative and commutative, with the empty definition ϵ as identity. Finally, we have a *scope extrusion* law, which states that the scope of a local definition may be extended dynamically over other operands of a parallel composition, provided this does not lead to clashes between names bound by the definition and free names of the terms that are brought in scope.

There is still just one reduction rule, and this rule is essentially the same as in the functional subset. The major difference is that now a rewrite step may involve sets of function applications, which are composed in parallel.

The laws of structural equivalence are necessary to bring parallel subterms which are “far apart” next to each other, so that they can match the join pattern of left-hand side. For instance, in the following example of semaphore synchronization two structural equivalences are necessary before rewrite steps can be performed.

Syntax:

Names	$a, b, c, \ldots, x, y, z$		
Identifiers	I, J	$=$	$x \mid I.x$
Terms	M, N	$=$	$\mathbf{def}\, D \;;\; M \mid I(\tilde{J}) \mid M \,\&\, N$
Definitions	D	$=$	$L = M \mid D, D \mid \epsilon$
Left-hand sides	L	$=$	$I(\tilde{y}) \mid L \,\&\, L$
Reduction contexts	R	$=$	$[\,] \mid \mathbf{def}\, D \;;\; R \mid R \,\&\, M \mid M \,\&\, R$

Structural Equivalence: α-renaming +

1. (&) on terms is associative and commutative:
$$M_1 \,\&\, M_2 \equiv M_2 \,\&\, M_1$$
$$M_1 \,\&\, (M_2 \,\&\, M_3) \equiv (M_1 \,\&\, M_2) \,\&\, M_3$$
2. $(,)$ on definitions is associative and commutative with ϵ as an identity:
$$D_1, D_2 \equiv D_2, D_1$$
$$D_1, (D_2, D_3) \equiv (D_1, D_2), D_3$$
$$D, \epsilon \equiv D$$
3. Scope extrusion:
$$(\mathbf{def}\, D \;;\; M) \,\&\, N \equiv \mathbf{def}\, D \;;\; (M \,\&\, N) \qquad \text{if } \mathrm{dn}(D) \cap \mathrm{fn}(N) = \emptyset.$$

Reduction:

$$\mathbf{def}\, D, L = M \;;\; R[\sigma L] \quad \rightarrow \quad \mathbf{def}\, D, L = M \;;\; R[\sigma M]$$

where σ is a substitution from names to identifiers with $\mathrm{dom}(\sigma) \subseteq \mathrm{ln}(L)$.

Fig. 5. Object-based join calculus

```
     def getLock(k) & releaseLock() = k();
     releaseLock() & (def k'() = f() & g(); getLock(k'))
≡    (by commutativity of &)
     def getLock(k) & releaseLock() = k();
     (def k'() = f() & g(); getLock(k')) & releaseLock()
≡    (by scope extrusion)
     def getLock(k) & releaseLock() = k();
     def k'() = f() & g(); getLock(k') & releaseLock()
→    def getLock(k) & releaseLock() = k(); def k'() = f() & g(); k'()
→    def getLock(k) & releaseLock() = k(); def k'() = f() & g(); f() & g()
```

6.3 Object-Based Calculus

Figure 5 presents the final stage of our progression, object-based join calculus. The only syntactical addition with respect to Figure 4 is that identifiers can now be qualified names. A qualified name I is either a simple name x or a qualified name followed by a period and a simple name. Qualified names can appear as

$$
\begin{array}{llll}
\mathrm{first}(x) & = x & \mathrm{ln}(I(x_1, \ldots, x_n)) & = \{x_1, \ldots, x_n\} \\
\mathrm{first}(I.f) & = \mathrm{first}(f) & \mathrm{ln}(L_1 \& L_2) & = \mathrm{ln}(L_1) \cup \mathrm{ln}(L_2) \\
 & & \mathrm{fn}(I(J_1, \ldots, J_n)) & = \{\mathrm{first}(I), \mathrm{first}(J_1), \ldots, \mathrm{first}(J_n)\} \\
\mathrm{dn}(I(\tilde{x})) & = \mathrm{first}(I) & \mathrm{fn}(\mathbf{def}\, D \;;\, M) & = (\mathrm{fn}(D) \cup \mathrm{fn}(M)) \backslash \mathrm{dn}(D) \\
\mathrm{dn}(L_1 \& L_2) & = \mathrm{dn}(L_1) \cup \mathrm{dn}(L_2) & \mathrm{fn}(M_1 \& M_2) & = \mathrm{fn}(M_1) \cup \mathrm{fn}(M_2) \\
\mathrm{dn}(D_1, D_2) & = \mathrm{dn}(D_1) \cup \mathrm{dn}(D_2) & \mathrm{fn}(L = M) & = \mathrm{dn}(L) \cup (\mathrm{fn}(M) \backslash \mathrm{ln}(L)) \\
 & & \mathrm{fn}(D_1, D_2) & = \mathrm{fn}(D_1) \cup \mathrm{fn}(D_2)
\end{array}
$$

Fig. 6. Local, defined, aiand free names

the operands of a function application and as defined function symbols in a definition.

Perhaps surprisingly, this is all that changes! The structural equivalences and reduction rules stay exactly as they were formulated for canonical join calculus. However, a bit of care is required in the definition of permissible renamings. For instance, consider the following object-based functional net:

def this.f(k) & g(x) = k(x) ; k'(this) & g(0)

In this net, both this and g can be consistently renamed. For instance, the following expression would be considered equivalent to the previous one:

def that.f(k) & h(x) = k(x) ; k'(that) & h(0)

On the other hand, the qualified function symbol f cannot be renamed without changing the meaning of the expression. For instance, renaming f to e would yield:

def this.e(k) & g(x) = k(x) ; k'(this) & g(0)

This is clearly different from the expression we started with. The new expression passes a record with an e field to the continuation function k', whereas the previous expressions passed a record with an f field.

Figure 6 reflects these observations in the definition of local, defined, and free names for object-based join calculus. Note that names occurring as field selectors are neither free in a term, nor are they defined or local. Hence α-renaming does not apply to record selectors.

The α-renaming rule is now formalized as follows. Let a *renaming* θ be a substitution from names to names which is injective when considered as a function from $\mathrm{dom}(\theta)$ (remember that $\mathrm{dom}(\theta) = \{x \mid \theta(x) \neq x\}$). Then,

$$\mathbf{def}\, D \;;\, M \equiv \mathbf{def}\, \theta D \;;\, \theta M$$

if θ is a renaming with $\mathrm{dom}(\theta) \subseteq \mathrm{dn}(D)$ and $\mathrm{codom}(\theta) \cap (\mathrm{fn}(D) \cup \mathrm{fn}(M)) = \emptyset$. Furthermore,

$$\mathbf{def}\, D, L = M \;;\, N \equiv \mathbf{def}\, D, \theta L = \theta M \;;\, N$$

Silk program:

```
    def newChannel = ( def this.read & this.write(x) = x ; this );
    val chan = newChannel;
    chan.read & chan.write(1)
```

Join calculus program and its reduction:

```
          def newChannel(k1) = (def this.read(k2) & this.write(x) = k2(x); k1(this));
          def k3(chan) = chan.read(k0) & chan.write(1);
          newChannel(k3)
    →
          def newChannel(k1) = (def this.read(k2) & this.write(x) = k2(x); k1(this));
          def k3(chan) = chan.read(k0) & chan.write(1);
          def this'.read(k2') & this'.write(x') = k2'(x');
          k3(this')
    →
          def newChannel(k1) = (def this.read(k2) & this.write(x) = k2(x); k1(this));
          def k3(chan) = chan.read(k0) & chan.write(1);
          def this'.read(k2') & this'.write(x') = k2'(x');
          this'.read(k0) & this'.write(1);
    →
          def newChannel(k1) = (def this.read(k2) & this.write(x) = k2(x); k1(this));
          def k3(chan) = chan.read(k0) & chan.write(1);
          def this'.read(k2') & this'.write(x') = k2'(x');
          k0(1)
```

Fig. 7. Reduction involving an asynchronous channel object

if θ is a renaming with $\mathrm{dom}(\theta) \subseteq \mathrm{ln}(L)$ and $\mathrm{codom}(\theta) \cap \mathrm{fn}(M) = \emptyset$.

The definitions of Figure 6 and the α-renaming rule apply as stated to all three versions of join calculus, not only to the final object-based version. When reduced to the simpler syntax of previous calculi, the new definitions are equivalent to the old ones.

As an example of object-based reduction consider the Silk program at the top of Figure 7. The program defines an asynchronous channel using function newChannel and then reads and writes that channel.

This program is not yet in the form mandated by join calculus since it uses a synchronous function and a **val** definition. We can map this program into join calculus by adding continuation functions which make control flow for function returns and value definitions explicit. The second half of Figure 7 shows how this program is coded in object-based join calculus and how it is reduced. Schemes which map from our programming notation to join calculus are further discussed in the next section.

7 Syntactic Abbreviations

Even the extended calculus discussed in the last section is a lot smaller than the Silk programming notation we have used in the preceding sections. This section fills the gap, by showing how Silk constructs which are not directly supported in object-based join calculus can be mapped into equivalent constructs which are supported.

Direct style An important difference between Silk and join calculus is that Silk has synchronous functions and **val** definitions which bind the results of synchronous function applications. To see the simplifications afforded by these additions, it suffices to compare the Silk program of Figure 7 with its join calculus counterpart. The join calculus version is much more cluttered because of the occurrence of the continuations k_i. Programs which make use of synchronous functions and value definitions are said to be in *direct style*, whereas programs that don't are said to be in *continuation passing style*. Join calculus supports only continuation passing style. To translate direct style programs into join calculus, we need a *continuation passing transform*. This transformation gives each synchronous function an additional argument which represents a continuation function, to which the result of the synchronous function is then passed.

The source language of the continuation passing transform is object-based join calculus extended with result expressions $(\mathsf{I}_1, ..., \mathsf{I}_n)$ and value definitions **val** $(\mathsf{x}_1, ..., \mathsf{x}_n) = \mathsf{M}\ ;\ \mathsf{N}$. Single names in results and value definitions are also included as they can be expressed as tuples of length 1.

For the sake of the following explanation, we assume different alphabets for synchronous and asynchronous functions. We let I^s range over identifiers whose final selector is a synchronous function, whereas I^a ranges over identifiers whose final selector is an asynchronous function. In practice, we can distinguish between synchronous and asynchronous functions also by means of a type system, so that different alphabets are not required.

Our continuation passing transform for terms is expressed as a function TC which takes a term in the source language and a name representing a continuation as arguments, mapping these to a term in object-based join calculus. It makes use of a helper function TD which maps a definition in the source language to one in object-based join calculus. To emphasize the distinction between the transforms TC, TD and their syntactic expression arguments, we write syntactic expressions in [[ ]] brackets. The transforms are defined as follows.

$$\begin{array}{llcl}
TC[[\ \textbf{val}\ (\tilde{\mathsf{x}}) = \mathsf{M}\ ;\ \mathsf{N} &]]\mathsf{k} & = & \textbf{def}\ \mathsf{k}'\ (\tilde{\mathsf{x}}) = TC[[\ \mathsf{N}\ ]]\mathsf{k}\ ;\ TC[[\ \mathsf{M}\ ]]\mathsf{k}' \\
TC[[\ (\mathsf{I}_1, ..., \mathsf{I}_n) &]]\mathsf{k} & = & \mathsf{k}(\mathsf{I}_1, ..., \mathsf{I}_n) \\
TC[[\ \mathsf{I}^s(\mathsf{J}_1, ..., \mathsf{J}_n) &]]\mathsf{k} & = & \mathsf{I}^s(\mathsf{J}_1, ..., \mathsf{J}_n, \mathsf{k}) \\
TC[[\ \mathsf{I}^a(\mathsf{J}_1, ..., \mathsf{J}_n) &]]\mathsf{k} & = & \mathsf{I}^a(\mathsf{J}_1, ..., \mathsf{J}_n) \\
TC[[\ \textbf{def}\ \mathsf{D}\ ;\ \mathsf{M} &]]\mathsf{k} & = & \textbf{def}\ TD[[\ \mathsf{D}\ ]]\ ;\ TC[[\ \mathsf{M}\ ]]\mathsf{k} \\
TD[[\ \mathsf{L} = \mathsf{M} &]] & = & TC[[\ \mathsf{L}\ ]]\mathsf{k}' = TC[[\ \mathsf{M}\ ]]\mathsf{k}' \\
TD[[\ \mathsf{D}, \mathsf{D}' &]] & = & TD[[\ \mathsf{D}\ ]], TD[[\ \mathsf{D}'\ ]] \\
TD[[\ \epsilon &]] & = & \epsilon
\end{array}$$

Here, the k' in the first equations for *TC* and *TD* represent fresh continuation names.

The original paper on join [15] defines a different contination passing transform. That transform allows several functions in a join pattern to carry results. Consequently, in the body of a function it has to be specified to which of the functions of a left hand side a result should be returned to. The advantage of this approach is that it simplifies the implementation of rendevouz situations like the synchronous channel of Section 5. The disadvantage is a more complex construct for function returns.

Structured Terms In Silk, the function part and arguments of a function application can be arbitrary terms, whereas join calculus admits only identifiers. Terms as function arguments can be expanded out by introducing names for intermediate results.

$$\mathsf{M}\ (\mathsf{N}_1,\ ....,\ \mathsf{N}_k) \quad\Rightarrow\quad \textbf{val}\ \mathsf{x} = \mathsf{M};\ \textbf{val}\ \mathsf{y}_1 = \mathsf{N}_1;\ ...\ \textbf{val}\ \mathsf{y}_k = \mathsf{N}_k;\ \mathsf{x}(\mathsf{y}_1,...,\mathsf{y}_k)$$

The resulting expression can be mapped into join calculus by applying the continuation passing transform *TC*. The same principle is also applied in other situations where structured terms appear yet only identifiers are supported. E.g.:

$$\begin{array}{lll} (\mathsf{M}_1,\ ....,\ \mathsf{M}_k) & \Rightarrow & \textbf{val}\ \mathsf{x}_1 = \mathsf{M}_1;\ ...\ \textbf{val}\ \mathsf{x}_k = \mathsf{M}_k;\ (\mathsf{x}_1,...,\mathsf{x}_k) \\ \mathsf{M.f} & \Rightarrow & \textbf{val}\ \mathsf{x} = \mathsf{M};\ \mathsf{x.f} \end{array}$$

We assume here that names in the expanded term which are not present in the original source term are fresh.

8 Conclusion and Related Work

The first five sections of this paper have shown how a large variety of program constructs can be modelled as functional nets. The last two sections have shown how functional nets themselves can be expressed in object-based join calculus. Taken together, these steps constitute a reductionistic approach, where a large body of notations and patterns of programs is to be distilled into a minimal kernel. The reduction to essentials is useful since it helps clarify the meaning of derived program constructs and the interactions between them.

Ever since the inception of Lisp [26] and Landin's ISWIM [25], functional programming has pioneered the idea of developing programming languages from calculi. Since then, there has been an extremely large body of work which aims to emulate the FP approach in a more general setting. One strand of work has devised extensions of lambda calculus with state [13,34,36,28,3] or non-determinism and concurrency [7,12,9]. Another strand of work has been designed concurrent functional languages [19,33,2] based on some other operational semantics. Landin's programme has also been repeated in the concurrent programming field, for instance with Occam and CSP [22], Pict [30] and π-calculus [27], or Oz and its kernel [35].

Our approach is closest to the work on join calculus [15,16,17,14]. Largely, functional nets as described here constitute a simplification and streamlining of the original treatment of join, with object-based join calculus and qualified definitions being the main innovation.

Acknowledgements

Many thanks to Matthias Zenger and Christoph Zenger, for designing several of the examples and suggesting numerous improvements.

References

1. F. Achermann, M. Lumpe, J.-G. Schneider, and O. Nierstrasz. Piccola - a small composition language. Submitted for Publication, available from `http://www.iam.unibe.ch/~scg/Research/Piccola`, 1999.
2. G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–69, 1997.
3. Z. Ariola and A. Sabry. Correctness of monadic state: An imperative call-by-need calculus. In *Proc. 25th ACM Symposium on Principles of Programming Languages*, pages 62–74, 1998.
4. A. Asperti and N. Bussi. Mobile petri nets. Technical Report UBLCS-96-10, University of Bolognia, May 1996.
5. H. P. Barendregt. *The Lambda Calculus: its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, revised edition, 1984.
6. G. Berry and G. Boudol. The chemical abstract machine. In *Proc. 17th ACM Symposium on Principles of Programming Languages*, pages 81–94, January 1990.
7. G. Boudol. Towards a lambda-calculus for concurrent and communicating systems. In J. Díaz and F. Orejas, editors, *Proceedings TAPSOFT '1989*, pages 149–161, New York, March 1989. Springer-Verlag. Lecture Notes in Computer Science 351.
8. G. Boudol. Asynchrony and the pi-calculus. Research Report 1702, INRIA, May 1992.
9. G. Boudol. The pi-calculus in direct style. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, pages 228–241, 1997.
10. A. Church. *The Calculi of Lambda-Conversion*, volume 6 of *Annals of Mathematics Studies*. Princeton University Press, second edition, 1951.
11. E. Crank and M. Felleisen. Parameter-passing and the lambda-calculus. In *Proc. 18th ACM Symposium on Principles of Programming Languages*, pages 233–244, January 1991.
12. U. de'Liguoro and A. Piperno. Non deterministic extensions of untyped λ-calculus. *Information and Computation*, 122(2):149–177, 1 Nov. 1995.
13. M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103:235–271, 1992.
14. F. L. Fessant. *The JoCaml reference manual.* INRIA Rocquencourt, 1998. Available from `http://join.inria.fr`.
15. C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proc. 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385, Jan. 1996.
16. C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *7th International Conference on Concurrency Theory (CONCUR'96)*, pages 406–421, Pisa, Italy, Aug. 26-29 1996. Springer-Verlag. LNCS 1119.

17. C. Fournet and L. Maranget. *The Join-Calculus Language.* INRIA Rocquencourt, 1997. Available from `http://join.inria.fr`.
18. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.
19. A. Giacalone, P. Mishra, and S. Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, April 1989.
20. P. B. Hansen. Structured multiprogramming. *Communications of the ACM*, 15(7):574–578, July 1972.
21. C. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 12(10), Oct. 74.
22. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, New Jersey, 1985.
23. K. Honda and N. Yoshida. On reduction-based process semantics. In *Proc. 13th Conf. on Foundations of Softawre Technology and Theoretical Computer Science*, pages 373–387, Dec. 1993.
24. K. Jensen. *Coloured Petri Nets. Basic Concepts.* EATCS Monographs on Theoretical Computer Science. Springer Verlag, 1992.
25. P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9:157–166, March 1966.
26. J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart, and I. L. Levin. *Lisp 1.5 Programmer's Manual.* MIT Press, 1969.
27. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100:1–77, 1992.
28. M. Odersky, D. Rabin, and P. Hudak. Call-by-name, assignment, and the lambda calculus. In *Proc. 20th ACM Symposium on Principles of Programming Languages*, pages 43–56, January 1993.
29. C. Petri. Kommunikation mit Automaten. Schriften des IIM 2, Institut für Instrumentelle Mathematik, Bonn, 1962. English translation: Technical Report RADC-TR-65-377, Vol. 1, Suppl. 1, Applied Data Research, Princeton, New Jersey, Contract AF 30 (602)-3324, 1966.
30. B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. Technical Report CSCI 476, Computer Science Department, Indiana University, 1997.
31. G. D. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.
32. W. Reisig. *Petri Nets*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.
33. J. H. Reppy. CML: A higher-order concurrent language. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 293–305, June 1991.
34. A. Sabry and J. Field. Reasoning about explicit and implicit representations of state. In *SIPL '93 ACM SIGPLAN Workshop on State in Programming Languages, Copenhagen, Denmark*, pages 17–30, June 1993. Yale University Research Report YALEU/DCS/RR-968.
35. G. Smolka, M. Henz, and J. Würtz. Object-oriented concurrent constraint programming in Oz. In P. van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming*, chapter 2, pages 29–48. The MIT Press, 1995.
36. V. Swarup, U. S. Reddy, and E. Ireland. Assignments for applicative languages. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, pages 192–214. Springer-Verlag, August 1991. Lecture Notes in Computer Science 523.

Faithful Translations between Polyvariant Flows and Polymorphic Types

Torben Amtoft[1] and Franklyn Turbak[2] *

[1] Boston University, Boston MA 02215, USA
tamtoft@bu.edu
[2] Wellesley College, Wellesley MA 02481, USA
fturbak@wellesley.edu

Abstract. Recent work has shown equivalences between various type systems and flow logics. Ideally, the translations upon which such equivalences are based should be *faithful* in the sense that information is not lost in round-trip translations from flows to types and back or from types to flows and back. Building on the work of Nielson & Nielson and of Palsberg & Pavlopoulou, we present the first faithful translations between a class of finitary polyvariant flow analyses and a type system supporting polymorphism in the form of intersection and union types. Additionally, our flow/type correspondence solves several open problems posed by Palsberg & Pavlopoulou: (1) it expresses call-string based polyvariance (such as k-CFA) as well as argument based polyvariance; (2) it enjoys a subject reduction property for flows as well as for types; and (3) it supports a flow-oriented perspective rather than a type-oriented one.

1 Introduction

Type systems and flow logic are two popular frameworks for specifying program analyses. While these frameworks seem rather different on the surface, both describe the "plumbing" of a program, and recent work has uncovered deep connections between them. For example, Palsberg and O'Keefe [PO95] demonstrated an equivalence between determining flow safety in the monovariant 0-CFA flow analysis and typability in a system with recursive types and subtyping [AC93]. Heintze showed equivalences between four restrictions of 0-CFA and four type systems parameterized by (1) subtyping and (2) recursive types [Hei95].

Because they merge flow information for all calls to a function, monovariant analyses are imprecise. Greater precision can be obtained via polyvariant analyses, in which functions can be analyzed in multiple abstract contexts. Examples of polyvariant analyses include call-string based approaches, such as k-CFA [Shi91,JW95,NN97], polymorphic splitting [WJ98], type-directed flow analysis [JWW97], and argument based polyvariance, such as Schmidt's analysis [Sch95] and Agesen's cartesian product analysis [Age95]. In terms of the

* Both authors were supported by NSF grant EIA–9806747. This work was conducted as part of the Church Project (http://www.cs.bu.edu/groups/church/).

G. Smolka (Ed.): ESOP/ETAPS 2000, LNCS 1782, pp. 26–40, 2000.

flow/type correspondence, several forms of flow polyvariance appear to correspond to type polymorphism expressed with intersection and union types [Ban97,WDMT97,DMTW97,PP99]. Intuitively, intersection types are finitary polymorphic types that model the multiple analyses for a given abstract closure, while union types are finitary existential types that model the merging of abstract values where flow paths join. Palsberg and Pavlopoulou (henceforth P&P) were the first to formalize this correspondence by demonstrating an equivalence between a class of flow analyses supporting argument based polyvariance and a type system with union and intersection types [PP99].

If type and flow systems encode similar information, translations between the two should be *faithful*, in the sense that round-trip translations from flow analyses to type derivations and back (or from type derivations to flow analyses and back) should not lose precision. Faithfulness formalizes the intuitive notion that a flow analysis and its corresponding type derivation contain the same information content. Interestingly, neither the translations of Palsberg and O'Keefe nor those of P&P are faithful. The lack of faithfulness in P&P is demonstrated by a simple example. Let $e = (\lambda^1\texttt{x.succ x})\,@\,((\lambda^2\texttt{y.y})\,@\,3)$, where we have labeled two program points of interest. Consider an initial monovariant flow analysis in which the only abstract closure reaching point 1 is $v_1 = (\lambda\texttt{x.succ x}, [\,])$ and the only one reaching point 2 is $v_2 = (\lambda\texttt{y.y}, [\,])$. The flow-to-type translation of P&P yields the expected type derivation:

$$\dfrac{\dfrac{\cdots}{[\,] \vdash \lambda^1\texttt{x.succ x} : \texttt{int} \to \texttt{int}} \qquad \dfrac{\dfrac{\cdots}{[\,] \vdash \lambda^2\texttt{y.y} : \texttt{int} \to \texttt{int}} \quad \cdots}{[\,] \vdash (\lambda^2\texttt{y.y})\,@\,3 : \texttt{int}}}{[\,] \vdash (\lambda^1\texttt{x.succ x})\,@\,((\lambda^2\texttt{y.y})\,@\,3) : \texttt{int}}$$

However, P&P's type-to-flow translation loses precision by merging into a single set all abstract closures associated with the same type in a given derivation. For the example derivation above, the type $\texttt{int} \to \texttt{int}$ translates back to the abstract closure set $V = \{v_1, v_2\}$, yielding a less precise flow analysis in which V flows to both points 1 and 2. In contrast, Heintze's translations are faithful. The undesirable merging in the above example is avoided by annotating function types with a label set indicating the source point of the function value. Thus, $\lambda^1\texttt{x.succ x}$ has type $\texttt{int} \overset{\{1\}}{\to} \texttt{int}$ while $\lambda^2\texttt{y.y}$ has type $\texttt{int} \overset{\{2\}}{\to} \texttt{int}$.

In this paper, we present the first faithful translations between a broad class of polyvariant flow analyses and a type system with polymorphism in the form of intersection and union types. The translations are faithful in the sense that a round-trip translation acts as the identity for canonical types/flows, and otherwise canonicalizes. In particular, our round-trip translation for types preserves non-recursive types that P&P may transform to recursive types. We achieve this result by adapting the translations of P&P to use a modified version of the flow analysis framework of Nielson and Nielson (henceforth N&N) [NN97]. As in Heintze's translations, annotations play a key role in the faithfulness of our translations: we (1) annotate flow values to indicate the sinks to which they flow, and (2) annotate union and intersection types with component labels. These annotations can be justified independently of the flow/type correspondence.

Additionally, our framework solves several open problems posed by P&P:

1. *Unifying P&P and N&N*: Whereas P&P's flow specification can readily handle only argument based polyvariance, N&N's flow specification can also express call-string based polyvariance. So our translations give the first type system corresponding to k-CFA analysis where $k \geq 1$.
2. *Subject reduction for flows*: We inherit from N&N's flow logic the property that flow information valid before a reduction step is still valid afterwards. In contrast, P&P's flow system does not have this property.
3. *Letting "flows have their way"*: P&P discuss mismatches between flow and type systems that imply the need to choose one perspective over the other when designing a translation between the two systems. P&P always let types "have their way"; for example they require analyses to be finitary and to analyze all closure bodies, even though they may be dead code. In contrast, our design also lets flows "have their way", in that our type system does not require all subexpressions to be analyzed.

Due to space limitations, the following presentation is necessarily somewhat dense. Please see the companion technical report [AT00] for a more detailed exposition with additional explanatory text, more examples, and proofs.

2 The Language

We consider a language whose core is λ-calculus with recursion:

$$\begin{array}{l} ue \in \mathbf{UnLabExpr} ::= z \mid \mu f.\lambda x.e \mid e @ e \mid c \mid \texttt{succ}\ e \mid \texttt{if0}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \ldots \\ e \in \mathbf{LabExpr} ::= ue^l \quad l \in \mathbf{Lab} \quad z \in \mathbf{Var} ::= x \mid f \quad x \in \mathbf{NVar} \quad f \in \mathbf{RVar} \end{array}$$

$\mu f.\lambda x.e$ denotes a function with parameter x which may call itself via f; $\lambda x.e$ is a shorthand for $\mu f.\lambda x.e$ where f does not occur in e. Recursive variables (ranged over by f) and non-recursive variables (ranged over by x) are distinct; z ranges over both. There are also integer constants c, the successor function, and the ability to test for zero. Other constructs might be added, e.g., `let`[1].

All subexpressions have integer labels. We often write labels on constructors (e.g., write $\lambda^l x.e$ for $(\lambda x.e)^l$ and $e_1 @_l e_2$ for $(e_1 @ e_2)^l$).

Example 1. The expression $\mathsf{P}_1 \equiv (\lambda^6 \mathtt{g}.((\mathtt{g}^3 @_2 \mathtt{g}^4) @_1 0^5)) @_0 (\lambda^8 \mathtt{x}.\mathtt{x}^7)$ shows the need for polyvariance: $\lambda^8 \mathtt{x}.\mathtt{x}^7$ is applied both to itself and to an integer.

Like N&N, but unlike P&P, we use an environment-based small step semantics. This requires incorporating N&N's `bind` and `close` constructs into our expression syntax. An expression not containing `bind` or `close` is said to be *pure*. Every abstraction body must be pure. A program P is a pure, closed expression where each label occurs at most once within P; thus each subexpression of P ($\in$ *SubExpr*$_P$) denotes a unique "position" within P.

[1] Let-polymorphism can be simulated by intersection types.

3 The Type System

Types are built from base types, function types, intersection types, and union types as follows (where **ITag** and **UTag** are unspecified):

$$\begin{array}{lll} t \in \mathbf{ElementaryType} ::= \texttt{int} \mid \bigwedge_{i\in I}\{K_i : u_i \to u'_i\} & & \\ u \in \mathbf{UnionType} ::= \bigvee_{i\in I}\{q_i : t_i\} & & \\ K \in \mathcal{P}(\mathbf{ITag}) & k \in \mathbf{ITag} & q \in \mathbf{UTag} \end{array}$$

Such grammars are usually interpreted inductively, but this one is to be viewed co-inductively. That is, types are regular (possibly infinite) trees formed according to the above specification. Two types are considered equal if their infinite unwindings are equal (modulo renaming of the index sets I).

An elementary type t is either an integer `int` or an intersection type of the form $\bigwedge_{i\in I}\{K_i : u_i \to u'_i\}$, where I is a (possibly empty) finite index set, each u_i and u'_i is a union type, and the K_i's, known as *I-tagsets*, are non-empty disjoint sets of *I-tags*. We write $dom(t)$ for $\cup_{i\in I} K_i$. Intuitively, if an expression e has the above intersection type then *for all* $i \in I$ it holds that the expression maps values of type u_i into values of type u'_i.

A union type u has the form $\bigvee_{i\in I}\{q_i : t_i\}$, where I is a (possibly empty) finite index set, each t_i is an elementary type, and the q_i are distinct *U-tags*. We write $dom(u)$ for $\cup_{i\in I}\{q_i\}$, and $u.q = t$ if there exists $i \in I$ such that $q = q_i$ and $t = t_i$. We assume that for all $i \in I$ it holds that $t_i = \texttt{int}$ iff $q_i = q_{\text{int}}$ where q_{int} is a distinguished U-tag. Intuitively, if an expression e has the above union type then *there exists* an $i \in I$ such that e has the elementary type t_i.

If $I = \{1 \cdots n\}$ $(n \geq 0)$, we write $\bigvee(q_1 : t_1, \cdots, q_n : t_n)$ for $\bigvee_{i\in I}\{t_i : q_i\}$ and write $\bigwedge(K_1 : u_1 \to u'_1, \cdots, K_n : u_n \to u'_n)$ for $\bigwedge_{i\in I}\{K_i : u_i \to u'_i\}$. We write u_{int} for $\bigvee(q_{\text{int}} : \texttt{int})$.

The type system is much as in P&P except for the presence of tags. These annotations serve as witnesses for existentials in the subtyping relation and play crucial roles in the faithfulness of our flow/type correspondence. U-tags track the "source" of each intersection type and help to avoid the precision-losing merging seen in P&P's type-to-flow translation (cf. Sect. 1). I-tagsets track the "sinks" of each arrow type and help to avoid unnecessary recursive types in the flow-to-type translation.

3.1 Subtyping

We define an ordering $\leq$ on union types and an ordering $\leq_\wedge$ on elementary types, where $u \leq u'$ means that u' is less precise than u and similarly for $\leq_\wedge$. To capture the intuition that something of type t_1 has one of the types t_1 or t_2, $\leq$ should satisfy $\bigvee(q_1 : t_1) \leq \bigvee(q_1 : t_1, q_2 : t_2)$. For $\leq_\wedge$, we want to capture the following intuition: a function that can be assigned both types $u_1 \to u'_1$ and $u_2 \to u'_2$ also (1) can be assigned one of them[2] and (2) can be assigned a function type

[2] I.e., for $i \in \{1, 2\}$, $\bigwedge(K_1 : u_1 \to u'_1, K_2 : u_2 \to u'_2) \leq_\wedge \bigwedge(K_i : u_i \to u'_i)$.

that "covers" both[3]. The following mutually recursive specification of $\leq$ and $\leq_\wedge$ formalizes these considerations:

$$\bigvee_{i\in I}\{q_i : t_i\} \leq \bigvee_{j\in J}\{q'_j : t'_j\}$$
iff for all $i \in I$ there exists $j \in J$ such that $q_i = q'_j$ and $t_i \leq_\wedge t'_j$

$\texttt{int} \leq_\wedge \texttt{int}$

$$\bigwedge_{i\in I}\{K_i : u_i \to u'_i\} \leq_\wedge \bigwedge_{j\in J}\{K'_j : u''_j \to u'''_j\}$$
iff for all $j \in J$ there exists $I_0 \subseteq I$ such that
$K'_j = \cup_{i\in I_0} K_i$ and $\forall i \in I_0.\ u'_i \leq u'''_j$ and
$\forall q \in \mathit{dom}(u''_j).\, \exists i \in I_0.\ q \in \mathit{dom}(u_i)$ and $u''_j.q \leq_\wedge u_i.q$.

Observe that if $t \leq_\wedge t'$, then $\mathit{dom}(t') \subseteq \mathit{dom}(t)$. The above specification is not yet a *definition* of $\leq$ and $\leq_\wedge$, since types may be infinite. However, it gives rise to a monotone functional on a complete lattice whose elements are pairs of relations; $\leq$ and $\leq_\wedge$ are then defined as the (components of) the *greatest* fixed point of this functional. Coinduction yields:

Lemma 1. *The relations $\leq$ and $\leq_\wedge$ are reflexive and transitive.*

Our subtyping relation differs from P&P's in several ways. The U-tags and I-tags serve as "witnesses" for the existential quantifiers present in the specification, reducing the need for search during type checking. Finally, our $\leq$ seems more natural that the P&P's $\leq_1$, which is not a congruence and in fact has the rather odd property that if $\vee(T_1, T_2) \leq_1 \vee(T_3, T_4)$ (with the T_i's all distinct), then either $\vee(T_1, T_2) \leq_1 T_3$ or $\vee(T_1, T_2) \leq_1 T_4$.

3.2 Typing Rules

A *typing* T for a program P is a tuple (P, IT_T, UT_T, D_T), where IT_T is a finite set of I-tags, UT_T is a finite set of U-tags, and D_T is a derivation of $[\,] \vdash P : u$ according to the inference rules given in Fig. 1. In a judgement $A \vdash e : u$, A is an environment with bindings of the form $[z \mapsto u]$; we require that all I-tags in D_T belong to IT_T and that all U-tags in D_T belong to UT_T.

Subtyping has been inlined in all of the rules to simplify the type/flow correspondence. The rules for function abstraction and function application are both instrumented with a "witness" that enables reconstructing the justification for applying the rule. In $[\mathsf{app}]^{w^@}$, the type of the operator is a (possibly empty) union, all components of which have the expected function type but the I-tagsets may differ; the app-*witness* $w^@$ is a partial mapping from $\mathit{dom}(u_1)$ that given q produces the corresponding I-tagset. In $[\mathsf{fun}]^{w^\lambda}$, the function types resulting from analyzing the body in several different environments are combined into an intersection type t. This is wrapped into a union type with an arbitrary U-tag q,

[3] I.e., $\bigwedge(K_1 : u_1 \to u'_1, K_2 : u_2 \to u'_2) \leq_\wedge \bigwedge(K_1 \cup K_2 : u_{12} \to u'_{12})$, where any value having one of the types u'_1 or u'_2 also has type u'_{12}, and where any value having type u_{12} also has one of the types u_1 or u_2.

$$
\begin{array}{lll}
[\mathsf{var}] & A \vdash z^l : u & \text{if } A(z) \le u \\[2ex]
[\mathsf{fun}]^{(q:t)} & \dfrac{\forall k \in K : A[f \mapsto u''_k, x \mapsto u_k] \vdash e : u'_k}{A \vdash \mu f.\lambda^l x.e : u} & \begin{array}{l}\text{if } t = \bigwedge_{k \in K}\{\{k\} : u_k \to u'_k\} \\ \wedge \bigvee(q : t) \le u \\ \wedge\ \forall k \in K.\ \bigvee(q : t) \le u''_k\end{array} \\[4ex]
[\mathsf{app}]^{w^@} & \dfrac{A \vdash e_1 : u_1 \quad A \vdash e_2 : u_2}{A \vdash e_1 @_l e_2 : u} & \text{if } \forall q \in \mathit{dom}(u_1).\, u_1.q \le_\wedge \bigwedge(w^@(q) : u_2 \to u) \\[3ex]
[\mathsf{con}] & A \vdash c^l : u & \text{if } u_{\mathrm{int}} \le u \\[2ex]
[\mathsf{suc}] & \dfrac{A \vdash e_1 : u_1}{A \vdash \mathtt{succ}^l\, e_1 : u} & \text{if } u_1 \le u_{\mathrm{int}} \le u \\[3ex]
[\mathsf{if}] & \dfrac{A \vdash e_0 : u_0 \quad A \vdash e_1 : u_1 \quad A \vdash e_2 : u_2}{A \vdash \mathtt{if0}^l\, e_0\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : u} & \text{if } u_0 \le u_{\mathrm{int}} \wedge u_1 \le u \wedge u_2 \le u
\end{array}
$$

Fig. 1. The typing rules

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{A_{\mathrm{g}} \vdash \mathsf{g}^3 : u_{\mathrm{x}} \quad A_{\mathrm{g}} \vdash \mathsf{g}^4 : u'_{\mathrm{x}}}{A_{\mathrm{g}} \vdash \mathsf{g}^3 @_2\, \mathsf{g}^4 : u'_{\mathrm{x}}} \quad A_{\mathrm{g}} \vdash 0^5 : u_{\mathrm{int}}
}{A_{\mathrm{g}} \vdash (\mathsf{g}^3 @_2\, \mathsf{g}^4) @_1\, 0^5 : u_{\mathrm{int}}}
}{[\,] \vdash \lambda^6 \mathsf{g}.((\mathsf{g}^3 @_2\, \mathsf{g}^4) @_1\, 0^5) : u_{\mathrm{g}}}
\quad
\dfrac{A_{\mathrm{x}} \vdash \mathtt{x}^7 : u_{\mathrm{int}} \quad A'_{\mathrm{x}} \vdash \mathtt{x}^7 : u'_{\mathrm{x}}}{[\,] \vdash \lambda^8 \mathtt{x}.\mathtt{x}^7 : u_{\mathrm{x}}}
}{[\,] \vdash (\lambda^6 \mathsf{g}.((\mathsf{g}^3 @_2\, \mathsf{g}^4) @_1\, 0^5)) @_0 (\lambda^8 \mathtt{x}.\mathtt{x}^7) : u_{\mathrm{int}}}
$$

Fig. 2. A derivation D_{T_1} for the program P_1 from Example 1.

which provides a way of keeping track of the origin of a function type (cf. Sects. 1 and 5). Accordingly, the fun-*witness* w^λ of this inference is the pair $(q : t)$. Note that K may be empty in which case the body is not analyzed.

Example 2. For the program P_1 from Ex. 1, we can construct a typing T_1 as follows: $IT_{\mathsf{T}_1} = \{0, 1, 2\}$, $UT_{\mathsf{T}_1} = \{q_{\mathrm{x}}, q_{\mathrm{g}}\}$, and D_{T_1} is as in Fig. 2, where

$$
\begin{array}{l}
u'_{\mathrm{x}} = \bigvee(q_{\mathrm{x}} : \bigwedge(\{1\} : u_{\mathrm{int}} \to u_{\mathrm{int}})) \\
u_{\mathrm{x}} = \bigvee(q_{\mathrm{x}} : \bigwedge(\{1\} : u_{\mathrm{int}} \to u_{\mathrm{int}}, \{2\} : u'_{\mathrm{x}} \to u'_{\mathrm{x}})) \\
u_{\mathrm{g}} = \bigvee(q_{\mathrm{g}} : \bigwedge(\{0\} : u_{\mathrm{x}} \to u_{\mathrm{int}})) \\
A_{\mathrm{g}} = [\mathsf{g} \mapsto u_{\mathrm{x}}] \qquad A_{\mathrm{x}} = [\mathtt{x} \mapsto u_{\mathrm{int}}] \qquad A'_{\mathrm{x}} = [\mathtt{x} \mapsto u'_{\mathrm{x}}]
\end{array}
$$

Note that $u_{\mathrm{x}} \le u'_{\mathrm{x}}$, and that $u_{\mathrm{x}}.q_{\mathrm{x}} \le_\wedge \bigwedge(\{2\} : u'_{\mathrm{x}} \to u'_{\mathrm{x}})$ so that $\{q_{\mathrm{x}} \mapsto \{2\}\}$ is indeed an app-witness for the inference at the top left of Fig. 2.

The type system in Fig. 1 can be augmented with rules for `bind` and `close` such that the resulting system satisfies a subject reduction property. The soundness of the type system follows from subject reduction, since "stuck" expressions (such as $7 @ 9$) are not typable.

In a typing T for P, for each e in $\mathit{SubExpr}_P$ there may be several judgements for e in D_T, due to the multiple analyses performed by [fun]. We assign to each judgement J for e in D_T an environment ke (its *address*) that for all applications of [fun] in the path from the root of D_T to J associates the bound variables with the branch taken. In D_{T_1} (Fig. 2), the judgement $A_{\mathtt{x}} \vdash \mathtt{x}^7 : u_{\text{int}}$ has address $[\mathtt{x} \mapsto 1]$ and the judgement $A'_{\mathtt{x}} \vdash \mathtt{x}^7 : u'_{\mathtt{x}}$ has address $[\mathtt{x} \mapsto 2]$.

The translation in Sect. 5 requires that a typing must be *uniform*, i.e., the following partial function A_T must be well-defined: $A_T(z, k) = u$ iff D_T contains a judgement of the form $A \vdash e : u'$ with address ke, where $ke(z) = k$ and $A(z) = u$. For T_1 we have, e.g., $A_{\mathsf{T}_1}(\mathtt{x}, 1) = u_{\text{int}}$ and $A_{\mathsf{T}_1}(\mathtt{x}, 2) = u'_{\mathtt{x}}$.

4 The Flow System

Our system for flow analysis has the form of a flow logic, in the style of N&N. A flow analysis F for program P is a tuple $(P, \mathit{Mem}_F, \mathcal{C}_F, \rho_F, \Phi_F)$, whose components are explained below (together with some auxiliary derived concepts).

Polyvariance is modeled by *mementoes*, where a memento ($m \in \mathit{Mem}_F$) represents a context for analyzing the body of a function. We shall assume that Mem_F is non-empty and *finite*; then all other entities occurring in F will also be finite. Each expression e is analyzed wrt. several different memento environments, where the entries of a memento environment ($me \in \mathit{MemEnv}_F$) take the form $[z \mapsto m]$ with m in Mem_F. Accordingly, a *flow configuration* ($\in \mathit{FlowConf}_F$) is a pair (e, me), where $FV(e) \subseteq \mathit{dom}(me)$.

The goal of the flow analysis is to associate a set of *flow values* to each configuration, where a flow value ($v \in \mathit{FlowVal}_F$) is either an integer Int or of the form (ac, M), where ac ($\in \mathit{AbsClos}_F$) is an *abstract closure* of the form (fn, me) with fn a function $\mu f.\lambda x.e$ and $FV(\mathit{fn}) \subseteq \mathit{dom}(me)$, and where $M \subseteq \mathit{Mem}_F$. The M component can be thought of a superset of the "sinks" of the abstract closure ac, i.e. the contexts in which it is going to be applied. Our flow values differ from N&N's in two respects: *(i)* they do not include the memento that corresponds to the point of definition; *(ii)* they do include the mementoes of use (the M component), in order to get a flow system that is almost isomorphic to the type system of Sect. 3. This extension does not make it harder to analyze an expression, since one might just let $M = \mathit{Mem}_F$ everywhere.

A *flow set* V ($\in \mathit{FlowSet}_F$) is a set of flow values, with the property that if $(ac, M_1) \in V$ and $(ac, M_2) \in V$ then $M_1 = M_2$. We define an ordering on $\mathit{FlowSet}_F$ by stipulating that $V_1 \leq_V V_2$ iff for all $v_1 \in V_1$ there exists $v_2 \in V_2$ such that $v_1 \leq_v v_2$, where the ordering $\leq_v$ on $\mathit{FlowVal}_F$ is defined by stipulating that Int $\leq_v$ Int and that $(ac, M_1) \leq_v (ac, M_2)$ iff $M_2 \subseteq M_1$. Note that if $V_1 \leq_V V_2$ then V_2 is obtained From V_1 by adding some "sources" and removing some "sinks (in a sense moving along a "flow path" from a source to a sink), so in that respect the ordering is similar to the type ordering in [WDMT97].

Φ_F is a partial mapping from $(\mathit{Labs}_P \times \mathit{MemEnv}_F) \times \mathit{AbsClos}_F$ to $\mathcal{P}(\mathit{Mem}_F)$, where Labs_P is the set of labels occurring in P. Intuitively, if the abstract closure ac in the context me is applied to an expression with label l, then $\Phi_F((l, me), ac)$ denotes the actual sinks of ac.

$\mathcal{C}_F$ is a mapping from $\mathit{Labs}_P \times \mathit{MemEnv}_F$ to $(\mathit{FlowSet}_F)_\perp$. Intuitively, if $\mathcal{C}_F(l, me) = V$ $(\neq \perp)$ and $\mathcal{C}_F$ is valid (defined below) for the flow configuration (ue^l, me) then all semantic values that ue^l may evaluate to in a semantic environment approximated by me can be approximated by the set V. Similarly, $\rho_F(z, m)$ approximates the set of semantic values to which z may be bound when analyzed in memento m.

Unlike N&N, we distinguish between $\mathcal{C}_F(l, me)$ being the empty set and being $\perp$. The latter means that no flow configuration (ue^l, me) is "reachable", and so there is no need to analyze it. The relation $\leq_V$ on $\mathit{FlowSet}_F$ is lifted to a relation $\leq_V$ on $\mathit{FlowSet}_{F\perp}$.

Example 3. For the program P_1 from Ex. 1, a flow analysis F_1 with $\mathit{Mem}_{\mathsf{F}_1} = \{0, 1, 2\}$ is given below. We have named some entities (note that $v_{\mathtt{x}} \leq_v v'_{\mathtt{x}}$):

$$\begin{array}{lll}
me_{\mathtt{g}} = [\mathtt{g} \mapsto 0] & ac_{\mathtt{g}} = (\lambda \mathtt{g}.\cdots, [\,]) & v_{\mathtt{g}} = (ac_{\mathtt{g}}, \{0\}) \\
me_{\mathtt{x}1} = [\mathtt{x} \mapsto 1] & ac_{\mathtt{x}} = (\lambda \mathtt{x}.\mathtt{x}^7, [\,]) & v'_{\mathtt{x}} = (ac_{\mathtt{x}}, \{1\}) \\
me_{\mathtt{x}2} = [\mathtt{x} \mapsto 2] & & v_{\mathtt{x}} = (ac_{\mathtt{x}}, \{1, 2\})
\end{array}$$

$\mathcal{C}_{\mathsf{F}_1}$ and ρ_{F_1} are given by the entries below (all other are $\perp$):

$$\begin{aligned}
\{v_{\mathtt{g}}\} &= \mathcal{C}_{\mathsf{F}_1}(6, [\,]) \\
\{\mathrm{Int}\} &= \rho_{\mathsf{F}_1}(\mathtt{x}, 1) = \mathcal{C}_{\mathsf{F}_1}(7, me_{\mathtt{x}1}) = \mathcal{C}_{\mathsf{F}_1}(5, me_{\mathtt{g}}) = \mathcal{C}_{\mathsf{F}_1}(1, me_{\mathtt{g}}) = \mathcal{C}_{\mathsf{F}_1}(0, [\,]) \\
\{v'_{\mathtt{x}}\} &= \rho_{\mathsf{F}_1}(\mathtt{x}, 2) = \mathcal{C}_{\mathsf{F}_1}(7, me_{\mathtt{x}2}) = \mathcal{C}_{\mathsf{F}_1}(4, me_{\mathtt{g}}) = \mathcal{C}_{\mathsf{F}_1}(2, me_{\mathtt{g}}) \\
\{v_{\mathtt{x}}\} &= \rho_{\mathsf{F}_1}(\mathtt{g}, 0) = \mathcal{C}_{\mathsf{F}_1}(3, me_{\mathtt{g}}) = \mathcal{C}_{\mathsf{F}_1}(8, [\,])
\end{aligned}$$

Thus $(\mathtt{g}^3 \mathbin{@_2} \mathtt{g}^4) \mathbin{@_1} 0^5$ is analyzed with $\mathtt{g}$ bound to 0, and $\mathtt{x}^7$ is analyzed twice: with $\mathtt{x}$ bound to 1 and with $\mathtt{x}$ bound to 2. Accordingly, Φ_{F_1} is given by

$$\Phi_{\mathsf{F}_1}((8, [\,]), ac_{\mathtt{g}}) = \{0\},\ \Phi_{\mathsf{F}_1}((5, me_{\mathtt{g}}), ac_{\mathtt{x}}) = \{1\},\ \Phi_{\mathsf{F}_1}((4, me_{\mathtt{g}}), ac_{\mathtt{x}}) = \{2\}.$$

4.1 Validity

Of course, not all flow analyses give a correct description of the program being analyzed. To formulate a notion of validity, we define a predicate $F \models^{me} e$ (to be read: F analyzes e correctly wrt. the memento environment me), with $(e, me) \in \mathit{FlowConf}_F$. The predicate must satisfy the specification in Fig. 3, which gives rise to a monotone functional on the complete lattice $\mathcal{P}(\mathit{FlowConf}_F)$; following the convincing argument of N&N, we define $F \models^{me} e$ as the *greatest* fixed point of this functional so as to be able to cope with recursive functions.

In [*fun*], we deviate from N&N by recording me, rather than the restriction of me to $FV(\mu f.\lambda x.e_0)$. As in P&P, this facilitates the translations to and from types. In [*app*], the set M corresponds to P&P's notion of *cover*, which in turn is needed to model the "cartesian product" algorithm of [Age95]. In N&N's framework, M is always a singleton $\{m\}$; in that case the condition "$\forall v \in \mathcal{C}_F(l_2, me).\ \ldots$ " amounts to the simpler "$\mathcal{C}_F(l_2, me) \leq_V \rho_F(x, m)$".

By structural induction in ue^l we see that if $F \models^{me} ue^l$ then $\mathcal{C}_F(l, me) \neq \perp$. We would also like the converse implication to hold:

$$
\begin{array}{ll}
[\mathit{var}] & F \models^{me} z^{l} \text{ iff } \bot \neq \rho_F(z, me(z)) \leq_V \mathcal{C}_F(l, me) \\[1ex]
[\mathit{fun}] & F \models^{me} \mu f.\lambda^{l} x.e_0 \text{ iff } \{((\mu f.\lambda x.e_0, me), \mathit{Mem}_F)\} \leq_V \mathcal{C}_F(l, me) \\[1ex]
[\mathit{app}] & F \models^{me} {ue_1}^{l_1} @_l {ue_2}^{l_2} \text{ iff} \\
& \qquad \mathcal{C}_F(l, me) \neq \bot \wedge F \models^{me} {ue_1}^{l_1} \wedge F \models^{me} {ue_2}^{l_2} \wedge \\
& \qquad \forall (ac_0, M_0) \in \mathcal{C}_F(l_1, me) \\
& \qquad\qquad \text{let } M = \Phi_F((l_2, me), ac_0) \text{ and } (\mu f.\lambda x.{ue_0}^{l_0}, me_0) = ac_0 \text{ in} \\
& \qquad\qquad M \subseteq M_0 \wedge \forall v \in \mathcal{C}_F(l_2, me).\, \exists m \in M.\, \{v\} \leq_V \rho_F(x, m) \wedge \\
& \qquad\qquad \forall m \in M\colon F \models^{me_0[f, x \mapsto m]} {ue_0}^{l_0} \wedge \\
& \qquad\qquad\qquad \mathcal{C}_F(l_0, me_0[f, x \mapsto m]) \leq_V \mathcal{C}_F(l, me) \wedge \\
& \qquad\qquad\qquad \rho_F(x, m) \neq \bot \wedge \{(ac_0, \mathit{Mem}_F)\} \leq_V \rho_F(f, m) \\[1ex]
[\mathit{con}] & F \models^{me} c^{l} \text{ iff Int} \in \mathcal{C}_F(l, me) \\[1ex]
[\mathit{suc}] & F \models^{me} \mathtt{succ}^{l}\, e_1 \text{ iff } F \models^{me} e_1 \wedge \text{Int} \in \mathcal{C}_F(l, me) \\[1ex]
[\mathit{if}] & F \models^{me} \mathtt{if0}^{l}\, e_0 \;\mathtt{then}\; {ue_1}^{l_1} \;\mathtt{else}\; {ue_2}^{l_2} \text{ iff} \\
& \qquad F \models^{me} e_0 \wedge F \models^{me} {ue_1}^{l_1} \wedge F \models^{me} {ue_2}^{l_2} \wedge \\
& \qquad \mathcal{C}_F(l_1, me) \leq_V \mathcal{C}_F(l, me) \wedge \mathcal{C}_F(l_2, me) \leq_V \mathcal{C}_F(l, me)
\end{array}
$$

Fig. 3. The flow logic

Definition 1. *Let a flow analysis F for P be given. We say that F is valid iff* (i) *$F \models^{[]} P$;* (ii) *whenever $e = ue^{l} \in \mathit{SubExpr}_P$ with $(e, me) \in \mathit{FlowConf}_F$ and $\mathcal{C}_F(l, me) \neq \bot$ then $F \models^{me} e$.*

Using techniques as in N&N, we can augment Fig. 3 with rules for `bind` and `close` and then prove a subject reduction property for flows which for closed E reads: if E reduces to E' in one evaluation step and $F \models^{[]} E$ then $F \models^{[]} E'$.

So far, even for badly behaved programs like $P = 7 @ 9$ it is possible (just as in N&N) to find a F for P such that F is valid. Since our type system rejects such programs, we would like to filter them out:

Definition 2. *Let a flow analysis F for P be given. We say that F is safe iff for all ue^{l} in $\mathit{SubExpr}_P$ and for all me it holds:* (i) *if $ue = {ue_1}^{l_1} @ e_2$ then $\mathit{Int} \notin \mathcal{C}_F(l_1, me)$;* (ii) *if $ue =$ `succ` ${ue_1}^{l_1}$ then $v \in \mathcal{C}_F(l_1, me)$ implies $v = \mathit{Int}$;* (iii) *if $ue =$ `if0` ${ue_0}^{l_0}$ `then` e_1 `else` e_2 then $v \in \mathcal{C}_F(l_0, me)$ implies $v = \mathit{Int}$.*

Example 4. Referring back to Example 3, it clearly holds that F_1 is safe, and it is easy (though a little cumbersome) to verify that F_1 is valid.

4.2 Taxonomy of Flow Analyses

Two common categories of flow analyses are the "call-string based" (e.g., [Shi91]) and the "argument-based" (e.g., [Sch95,Age95]). Our *descriptive* framework can model both approaches (which can be "mixed", as in [NN99]).

A flow analysis F for P such that F is valid is in $\mathit{CallString}^{P}_{\beta}$, where β is a mapping from $\mathit{Labs}_P \times \mathit{MemEnv}_F$ into Mem_F, iff whenever $\Phi_F((l_2, me), ac)$ is defined it equals $\{\beta(l, me)\}$ where l is such that[4] $e_1 \mathbin{@_l} ue_2{}^{l_2} \in \mathit{SubExpr}_P$. All k-CFA analyses fit into this category: for 0-CFA we take $\mathit{Mem}_F = \{\bullet\}$ and $\beta(l, me) = \bullet$; for 1-CFA we take $\mathit{Mem}_F = \mathit{Labs}_P$ and $\beta(l, me) = l$; and for 2-CFA (the generalization to $k > 2$ is immediate) we take $\mathit{Mem}_F = \mathit{Labs}_P \cup (\mathit{Labs}_P \times \mathit{Labs}_P)$ and define $\beta(l, me)$ as follows: let it be l if $me = [\,]$, and let it be (l, l_1) if me takes the form $me'[z \mapsto m]$ with m either l_1 or (l_1, l_2).

A flow analysis F for P such that F is valid is in $\mathit{ArgBased}_\alpha^P$ iff for all non-recursive variables x and mementoes m it holds that whenever $\rho_F(x, m) \neq \bot$ then $\epsilon_V(\rho_F(x, m)) = \alpha(m)$ where ϵ_V removes the M component of a flow value. For this kind of analysis, a memento m essentially denotes a set of abstract closures. To more precisely capture specific argument-based analyses, such as [Age95] or the type-directed approach of [JWW97], we may impose further demands on α.

Example 5. The flow analysis $\mathsf{F_1}$ is a 1-CFA and also in $\mathit{ArgBased}_\alpha^{\mathsf{P_1}}$, with $\alpha(0) = \alpha(2) = \{ac_{\mathrm{x}}\}$ and $\alpha(1) = \{\mathrm{Int}\}$.

Given a program P, it turns out that for all β the class $\mathit{CallString}_\beta^P$, and for certain kinds of α also the class $\mathit{ArgBased}_\alpha^P$, contains a least (i.e., most precise) flow analysis; here the ordering on flow analyses is defined pointwise[5] on $\mathcal{C}_F$, ρ_F and Φ_F. This is much as in N&N where for all total and deterministic "instantiators" the corresponding class of analyses contains a least element, something we cannot hope for since we allow Φ_F to return a non-singleton.

4.3 Reachability

For a flow analysis F, some entries may be garbage. To see an example of this, suppose that $\mu f.\lambda x.ue^l$ in $\mathit{SubExpr}_P$, and suppose that $\rho_F(x, m) = \bot$ for all $m \in \mathit{Mem}_F$. From this we infer that the above function is never calle so for all me the value of $\mathcal{C}_F(l, me)$ is uninteresting. It may therefore be replaced by $\bot$, something which is in fact achieved by the roundtrip described in Sect. 7.1.

To formalize a notion of reachability we introduce a set Reach_P^F that is intended to encompass[6] all entries of $\mathcal{C}_F$ and ρ_F that are "reachable" from the root of P. Let $\mathit{Analyzes}_m^F(\mu f.\lambda x.ue_0{}^{l_0}, me)$ be a shorthand for $\mathcal{C}_F(l_0, me[f, x \mapsto m]) \neq \bot$ and $\rho_F(x, m) \neq \bot$ and $\{((\mu f.\lambda x.ue_0{}^{l_0}, me), \mathit{Mem}_F)\} \leq_V \rho_F(f, m)$. We define Reach_P^F as the least set satisfying:

[prg] $(P, [\,]) \in \mathit{Reach}_P^F$

[fun] $\Big((\mu f.\lambda^l x.ue_0{}^{l_0}, me) \in \mathit{Reach}_P^F \,\wedge\, \mathit{Analyzes}_m^F(\mu f.\lambda x.ue_0{}^{l_0}, me))\Big) \Rightarrow$
$\{(ue_0{}^{l_0}, me[f, x \mapsto m]),\ (x, m),\ (f, m)\} \subseteq \mathit{Reach}_P^F$

[app] $(e_1 \mathbin{@_l} e_2, me) \in \mathit{Reach}_P^F \Rightarrow \{(e_1, me),\ (e_2, me)\} \subseteq \mathit{Reach}_P^F$

[4] It is tempting to write "$\Phi_F((l, me), ac_0)$" in Fig. 3 (thus replacing l_2 by l), but then subject reduction for flows would not hold.

[5] Unlike [JWW97], we do not compare analyses with different sets of mementoes.

[6] This is somewhat similar to the reachability predicate of [GNN97].

[suc] $(\texttt{succ}^l\ e_1, me) \in \mathit{Reach}_P^F \Rightarrow (e_1, me) \in \mathit{Reach}_P^F$

[if] $(\texttt{if0}^l\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2, me) \in \mathit{Reach}_P^F \Rightarrow$
$\{(e_0, me),\ (e_1, me),\ (e_2, me)\} \subseteq \mathit{Reach}_P^F$

Example 6. It is easy to verify that for $ue^l \in \mathit{SubExpr}_{\mathsf{P}_1}$ it holds that $\mathcal{C}_{\mathsf{F}_1}(, l)me \neq \bot$ iff $(ue^l, me) \in \mathit{Reach}_{\mathsf{P}_1}^{\mathsf{F}_1}$, and that $\rho_{\mathsf{F}_1}(z, m) \neq \bot$ iff $(z, m) \in \mathit{Reach}_{\mathsf{P}_1}^{\mathsf{F}_1}$.

Lemma 2. *Let F be a flow analysis for P such that F is valid. If $(ue^l, me) \in \mathit{Reach}_P^F$ then* (i) *$\mathcal{C}_F(l, me) \neq \bot$ and* (ii) *whenever $(z \mapsto m) \in me$ then $(z, m) \in \mathit{Reach}_P^F$ holds. Also, if $(z, m) \in \mathit{Reach}_P^F$ then $\rho_F(z, m) \neq \bot$.*

5 Translating Types to Flows

Let a uniform typing T for a program P be given. We now demonstrate how to construct a corresponding flow analysis $F = \mathcal{F}(T)$ such that F is valid and safe. First define Mem_F as IT_T; note that then an address can serve as a memento environment. Next we define a function $\mathcal{F}_T$ that translates from UTyp_T, that is the union types that can be built using IT_T and UT_T, into $\mathit{FlowSet}_F$:

$$\begin{aligned}
&\mathcal{F}_T(\textstyle\bigvee_{i \in I}\{q_i : t_i\}) = \\
&\quad \{((\mu f.\lambda x.e, me), M) \mid \exists i \in I \text{ with } M = \mathit{dom}(t_i)\text{:} \\
&\qquad \text{a judgement for } \mu f.\lambda^l x.e \text{ occurs in } D_T \text{ with address } me \\
&\qquad\quad \text{and is justified by } [\mathsf{fun}]^{(q_i:t)} \text{ where } t \leq_\wedge t_i\} \\
&\quad \cup\ (\text{if } \exists i. \text{ such that } q_i = q_{\text{int}} \text{ then } \{\text{Int}\} \text{ else } \emptyset)
\end{aligned}$$

The idea behind the translation is that $\mathcal{F}_T(u)$ should contain all the closures that are "sources" of elementary types in u; it is easy to trace such closures thanks to the presence of U-tags. The condition $t \leq_\wedge t_i$ is needed as a "sanity check", quite similar to the "trimming" performed in [Hei95], to guard against the possibility that two unrelated entities in D_T incidentally have used the same U-tag q_i. As the types of P&P do not contain fun-witnesses, their translation has to rely solely on this sanity check (at the cost of precision, cf. Sect. 1).

Lemma 3. *The function $\mathcal{F}_T$ is monotone.*

Definition 3. *With T a typing for P, the flow analysis $F = \mathcal{F}(T)$ is given by $(P, \mathit{IT}_T, \mathcal{C}_F, \rho_F, \Phi_F)$, where $\mathcal{C}_F$, ρ_F, and Φ_F are defined below:*

$\mathcal{C}_F(l, me) = \mathcal{F}_T(u)$ *iff D_T contains a judgement $A \vdash ue^l : u$ with address me*

$\rho_F(z, m) = \mathcal{F}_T(u)$ *iff $u = A_T(z, m)$*

$\Phi_F((l_2, me), (\mu f.\lambda x.e_0, me')) = M$ *iff there exists q such that D_T contains*
a judgement for $\mu f.\lambda x.e_0$ at me' derived by $[\mathsf{fun}]^{(q:t)}$,
a judgement for $e_1 \,@\, {ue_2}^{l_2}$ at me derived by $[\mathsf{app}]^{w^@}$ where $w^@(q) = M$.

Example 7. With terminology as in Examples 2 and 3, it is easy to check that $\mathcal{F}_{\mathsf{T}_1}(u'_{\mathtt{x}}) = \{v'_{\mathtt{x}}\}$ and that $\mathcal{F}_{\mathsf{T}_1}(u_{\mathtt{x}}) = \{v_{\mathtt{x}}\}$, and that $\mathsf{F}_1 = \mathcal{F}(\mathsf{T}_1)$.

We have the following result, where the proof that F is valid is by coinduction.

Theorem 1. *With T a uniform typing for P, for $F = \mathcal{F}(T)$ it holds that*

- *F is valid and safe*
- *$(ue^l, me) \in \mathit{Reach}_P^F$ iff $\mathcal{C}_F(l, me) \neq \bot$ (for $ue \in \mathit{SubExpr}_P$)*
- *$(z, m) \in \mathit{Reach}_P^F$ iff $\rho_F(z, m) \neq \bot$*

6 Translating Flows to Types

Let a flow analysis F for a program P be given, and assume that F is valid and safe. We now demonstrate how to construct a corresponding uniform typing $T = \mathcal{T}(F)$. First we define IT_T as Mem_F and UT_T as $\mathit{AbsClos}_F \cup \{q_{\text{int}}\}$. Next we define a function $\mathcal{T}_F$ that translates from $\mathit{FlowSet}_F$ into UTyp_T; inspired by P&P (though the setting is somewhat different) we stipulate:

$\mathcal{T}_F(V) = \bigvee_{v \in V} \{q_v : t_v\}$ where
  if $v = \text{Int}$ then $q_v = q_{\text{int}}$ and $t_v = \texttt{int}$
  if $v = (ac, M)$ with $ac = (\mu f.\lambda x.{e_0}^{l_0}, me)$
    then $q_v = ac$
    and $t_v = \bigwedge_{m \in M_0} \{\{m\} : \mathcal{T}_F(\rho_F(x, m)) \to \mathcal{T}_F(\mathcal{C}_F(l_0, me[f, x \mapsto m]))\}$
      where $M_0 = \{m \in M \mid \mathit{Analyzes}_m^F(ac)\}$.

The above definition determines a unique union type $\mathcal{T}_F(V)$, since recursion is "beneath a constructor" and since $\mathit{FlowSet}_F$ is finite (ensuring regularity).

Example 8. With terminology as in Examples 2 and 3, it is easy to see—provided that $q_{\mathtt{x}}$ is considered another name for $ac_{\mathtt{x}}$—first that $\mathcal{T}_{\mathsf{F}_1}(\{v'_{\mathtt{x}}\}) = u'_{\mathtt{x}}$, and then that $\mathcal{T}_{\mathsf{F}_1}(\{v_{\mathtt{x}}\}) = u_{\mathtt{x}}$ since $\mathcal{T}_{\mathsf{F}_1}(\{v_{\mathtt{x}}\}).q_{\mathtt{x}}$ can be found as

$$\begin{aligned}
&\bigwedge(\{1\} : \mathcal{T}_{\mathsf{F}_1}(\rho_{\mathsf{F}_1}(\mathtt{x}, 1)) \to \mathcal{T}_{\mathsf{F}_1}(\mathcal{C}_{\mathsf{F}_1}(7, me_{\mathtt{x}1})), \{2\} : \mathcal{T}_{\mathsf{F}_1}(\rho_{\mathsf{F}_1}(\mathtt{x}, 2)) \to \mathcal{T}_{\mathsf{F}_1}(\mathcal{C}_{\mathsf{F}_1}(7, me_{\mathtt{x}2})))\\
= &\bigwedge(\{1\} : \mathcal{T}_{\mathsf{F}_1}(\{\text{Int}\}) \to \mathcal{T}_{\mathsf{F}_1}(\{\text{Int}\}), \{2\} : \mathcal{T}_{\mathsf{F}_1}(\{v'_{\mathtt{x}}\}) \to \mathcal{T}_{\mathsf{F}_1}(\{v'_{\mathtt{x}}\}))\\
= &\bigwedge(\{1\} : u_{\text{int}} \to u_{\text{int}}, \{2\} : u'_{\mathtt{x}} \to u'_{\mathtt{x}}).
\end{aligned}$$

Note that without the M component in a flow value (ac, M), $v_{\mathtt{x}}$ would equal $v'_{\mathtt{x}}$ causing $\mathcal{T}_{\mathsf{F}_1}(\{v_{\mathtt{x}}\})$ to be an infinite type (as in P&P).

Lemma 4. *The function $\mathcal{T}_F$ is monotone.*

For z and m such that $(z, m) \in \mathit{Reach}_P^F$, we define $\mathcal{T}_F^\rho(z, m)$ as $\mathcal{T}_F(\rho_F(z, m))$ (by Lemma 2 this is well-defined). And for $e = ue^l$ and me such that $(e, me) \in \mathit{Reach}_P^F$, we construct a judgement $\mathcal{T}_F^J(e, me)$ as

$$\mathcal{T}_F^A(me) \vdash e : \mathcal{T}_F(\mathcal{C}_F(l, me))$$

where $\mathcal{T}_F^A(me)$ is defined recursively by $\mathcal{T}_F^A([\,]) = [\,]$ and $\mathcal{T}_F^A(me[z \mapsto m]) = \mathcal{T}_F^A(me)[z \mapsto \mathcal{T}_F^\rho(z, m)]$ (by Lemma 2 also this is well-defined).

Definition 4. *With F a flow analysis for P, the typing $T = \mathcal{T}(F)$ is given by $(P, Mem_F, AbsClos_F \cup \{q_{\text{int}}\}, D_T)$, where D_T is defined by stipulating that whenever (e, me) is in $Reach_P^F$ then D_T contains $\mathcal{T}_F^J(e, me)$, and that $\mathcal{T}_F^J(e', me')$ is a premise of $\mathcal{T}_F^J(e, me)$ iff $(e, me) \in Reach_P^F$ is among the immediate conditions (cf. the definition of $Reach_P^F$) for $(e', me') \in Reach_P^F$.*

Example 9. It is easy to check that $\mathsf{T}_1 = \mathcal{T}(\mathsf{F}_1)$, modulo renaming of the U-tags.

Clearly D_T is a tree-formed derivation, and $\mathcal{T}_F^J(e, me)$ has address me in D_T. We must of course also prove that all judgements in D_T are in fact derivable from their premises using the inference rules in Fig. 1:

Theorem 2. *If F is valid and safe then $T = \mathcal{T}(F)$ as constructed by Definition 4 is a uniform typing for P. The derivation D_T has the following properties:*

- *if D_T contains at address me a judgement for $\mu f.\lambda x.e$, it is derived using $[\mathsf{fun}]^{w^\lambda}$ where $w^\lambda = (ac : (\mathcal{T}_F(\{(ac, Mem_F)\})).ac)$ with $ac = (\mu f.\lambda x.e, me)$;*
- *if D_T contains at address me a judgement for $e_1 @ ue_2{}^{l_2}$ with the leftmost premise of the form $A \vdash e_1 : u_1$, then it is derived using $[\mathsf{app}]^{w^@}$ where for all $q \in dom(u_1)$ it holds that $w^@(q) = \Phi_F((l_2, me), q)$.*

7 Round Trips

Next consider the "round-trip" translations $\mathcal{F} \circ \mathcal{T}$ (from flows to types and back) and $\mathcal{T} \circ \mathcal{F}$ (from types to flows and back). Both roundtrips are idempotent[7]: they act as the identity on "canonical" elements, and otherwise "canonicalize".

Example 10. Exs. 7 and 9 show that $\mathcal{F} \circ \mathcal{T}$ is the identity on F_1 and that $\mathcal{T} \circ \mathcal{F}$ is the identity (modulo renaming of U-tags) on T_1. In particular $\mathcal{T} \circ \mathcal{F}$ does not necessarily introduce infinite types, thus solving an open problem in P&P.

7.1 Round Trips from the Flow World

$\mathcal{F} \circ \mathcal{T}$ filters out everything not reachable, and acts as the identity ever after.

Theorem 3. *Assume that F is valid and safe for a program P, and let $F' = \mathcal{F}(\mathcal{T}(F))$. Then F' is valid and safe for P with $Mem_{F'} = Mem_F$, $Reach_P^{F'} = Reach_P^F$, and $\mathcal{C}_{F'}(l, me) \neq \bot$ iff $\left(\mathcal{C}_F(l, me) \neq \bot \text{ and } (ue^l, me) \in Reach_P^F\right)$ in which case $\mathcal{C}_{F'}(l, me) = \text{filter}_P^F(\mathcal{C}_F(l, me))$ where $\text{filter}_P^F(V)$ is given by*

$$\{(ac, M') \mid (ac, M) \in V \text{ and } (\mu f.\lambda x.e_0, me_0) \in Reach_P^F \text{ where}$$
$$ac = (\mu f.\lambda x.e_0, me_0) \text{ and } M' = \{m \in M \mid (e_0, me_0[f, x \mapsto m]) \in Reach_P^F\}$$
$$\cup (\text{if } Int \in V \text{ then } \{Int\} \text{ else } \emptyset).$$

Finally, if $F'' = \mathcal{F}(\mathcal{T}(F'))$ then $F'' = F'$.

[7] However, $\mathcal{T}(\mathcal{F}(\mathcal{T}(F))) = \mathcal{T}(F)$ does in general not hold.

Clearly everything not reachable may be considered “junk”. However, simple examples demonstrate that some junk is reachable and is hence not removed by $\mathcal{F} \circ \mathcal{T}$. That our flow/type correspondence can faithfully encode such imprecisions illustrates the power of our framework.

7.2 Round Trips from the Type World

The canonical typings are the ones that are *strongly consistent*:

Definition 5. *A typing T is strongly consistent iff for all u that occur in D_T and for all $q \in dom(u)$ with $q \neq q_{\mathrm{int}}$ the following holds: D_T contains exactly one judgement derived by an application of $[\mathsf{fun}]^{w^\lambda}$ with w^λ taking the form $(q : t)$, and this t satisfies $t \leq^c_\wedge u.q$. Here $\leq^c_\wedge$ is a subrelation of $\leq_\wedge$, defined by stipulating that $\mathtt{int} \leq^c_\wedge \mathtt{int}$ and that $\bigwedge_{i \in I}\{K_i : u_i \to u'_i\} \leq^c_\wedge \bigwedge_{i \in I_0}\{K_i : u_i \to u'_i\}$ iff $I_0 \subseteq I$.*

Theorem 4. *Assume that T is a uniform typing for a program P, and let $T' = \mathcal{T}(\mathcal{F}(T))$. Then T' is a uniform typing for P with $IT_{T'} = IT_T$, and*

- *$D_{T'}$ contains a judgement for e with address ke iff D_T contains a judgement for e with address ke (i.e., the two derivations have the same shape);*
- *$D_{T'}$ is strongly consistent;*
- *if D_T is strongly consistent then $D_{T'} = D_T$ (modulo renaming of U-tags).*

Example 11. Let T be the typing[8] of the motivating example put forward in Sect. 1. Then T is not strongly consistent, but $T' = \mathcal{T}(\mathcal{F}(T))$ is: the two fun-witnesses occurring in $D_{T'}$ are of the form $(q_{\mathtt{x}} : u_{\mathrm{int}} \to u_{\mathrm{int}})$ and $(q_{\mathtt{y}} : u_{\mathrm{int}} \to u_{\mathrm{int}})$. Nevertheless, T' is still imprecise: both function abstractions are assigned the union type $\bigvee(q_{\mathtt{x}} : u_{\mathrm{int}} \to u_{\mathrm{int}}, q_{\mathtt{y}} : u_{\mathrm{int}} \to u_{\mathrm{int}})$.

8 Discussion

Our flow system follows the lines of N&N, generalizing some features while omitting others (such as polymorphic splitting [WJ98], left for future work). That it has substantial descriptive power is indicated by the fact that it encompasses both argument-based and call-string based polyvariance. In particular, the flow analysis framework of P&P can be encoded into our framework. Unlike P&P, our flow logic has a subject reduction property, inherited from the N&N approach.

The generality of our type system is less clear. The annotation with tags gives rise to intersection and union types that are not associative, commutative, or idempotent (ACI). This stands in contrast to the ACI types of P&P, but is similar to the non-ACI intersection and union types of CIL, the intermediate language of an experimental compiler that integrates flow information into the type system [WDMT97,DMTW97]. Indeed, a key motivation of this work was to formalize the encoding of various flow analyses in the CIL type system. Developing a translation between the the type system of this paper and CIL is our next goal.

[8] We convert it to our framework by substituting u_{int} for $\mathtt{int}$ and by substituting $\bigvee(q_\bullet : \bigwedge(\{\bullet\} : u_{\mathrm{int}} \to u_{\mathrm{int}}))$ for $\mathtt{int} \to \mathtt{int}$.

References

AC93. R. Amadio and L. Cardelli. Subtyping recursive types. *ACM Trans. on Prog. Langs. and Systs.*, 15(4):575–631, 1993.

Age95. O. Agesen. The cartesian product algorithm. In *Proceedings of ECOOP'95, Seventh European Conference on Object-Oriented Programming*, vol. 952, pp. 2–26. Springer-Verlag, 1995.

AT00. T. Amtoft and F. Turbak. Faithful translations between polyvariant flows and polymorphic types. Technical Report BUCS-TR-2000-01, Comp. Sci. Dept., Boston Univ., 2000.

Ban97. A. Banerjee. A modular, polyvariant, and type-based closure analysis. In ICFP '97 [ICFP97].

DMTW97. A. Dimock, R. Muller, F. Turbak, and J. B. Wells. Strongly typed flow-directed representation transformations. In ICFP '97 [ICFP97], pp. 11–24.

GNN97. K. L. S. Gasser, F. Nielson, and H. R. Nielson. Systematic realisation of control flow analyses for CML. In ICFP '97 [ICFP97], pp. 38–51.

Hei95. N. Heintze. Control-flow analysis and type systems. In SAS '95 [SAS95], pp. 189–206.

ICFP97. *Proc. 1997 Int'l Conf. Functional Programming*, 1997.

JW95. S. Jagannathan and S. Weeks. A unified treatment of flow analysis in higher-order languages. In *Conf. Rec. 22nd Ann. ACM Symp. Princ. of Prog. Langs.*, pp. 393–407, 1995.

JWW97. S. Jagannathan, S. Weeks, and A. Wright. Type-directed flow analysis for typed intermediate languages. In *Proc. 4th Int'l Static Analysis Symp.*, vol. 1302 of *LNCS*. Springer-Verlag, 1997.

NN97. F. Nielson and H. R. Nielson. Infinitary control flow analysis: A collecting semantics for closure analysis. In *Conf. Rec. POPL '97: 24th ACM Symp. Princ. of Prog. Langs.*, pp. 332–345, 1997.

NN99. F. Nielson and H. R. Nielson. Interprocedural control flow analysis. In *Proc. European Symp. on Programming*, vol. 1576 of *LNCS*, pp. 20–39. Springer-Verlag, 1999.

PO95. J. Palsberg and P. O'Keefe. A type system equivalent to flow analysis. *ACM Trans. on Prog. Langs. and Systs.*, 17(4):576–599, 1995.

PP98. J. Palsberg and C. Pavlopoulou. From polyvariant flow information to intersection and union types. In *Conf. Rec. POPL '98: 25th ACM Symp. Princ. of Prog. Langs.*, pp. 197–208, 1998. Superseded by [PP99].

PP99. J. Palsberg and C. Pavlopoulou. From polyvariant flow information to intersection and union types. A substantially revised version of [PP98]. Available at http://www.cs.purdue.edu/homes/palsberg/paper /popl98.ps.gz., Feb. 1999.

SAS95. *Proc. 2nd Int'l Static Analysis Symp.*, vol. 983 of *LNCS*, 1995.

Sch95. D. Schmidt. Natural-semantics-based abstract interpretation. In SAS '95 [SAS95], pp. 1–18.

Shi91. O. Shivers. *Control Flow Analysis of Higher Order Languages.* PhD thesis, Carnegie Mellon University, 1991.

WDMT97. J. B. Wells, A. Dimock, R. Muller, and F. Turbak. A typed intermediate language for flow-directed compilation. In *Proc. 7th Int'l Joint Conf. Theory & Practice of Software Development*, pp. 757–771, 1997.

WJ98. A. Wright and S. Jagannathan. Polymorphic splitting: An effective polyvariant flow analysis. *ACM Trans. on Prog. Langs. and Systs.*, 20:166–207, 1998.

On the Expressiveness of Event Notification in Data-Driven Coordination Languages*

Nadia Busi and Gianluigi Zavattaro

Dipartimento di Scienze dell'Informazione, Università di Bologna,
Mura Anteo Zamboni 7, I-40127 Bologna, Italy.
busi,zavattar@cs.unibo.it

Abstract. JavaSpaces and TSpaces are two coordination middlewares for distributed Java programming recently proposed by Sun and IBM, respectively. They are both inspired by the Linda coordination model: processes interact via the emission (*out*), consumption (*in*) and the test for absence (*inp*) of data inside a shared repository. The most interesting improvement introduced by these new products is the *event notification* mechanism (*notify*): a process can register interest in the incoming arrivals of a particular kind of data, and then receive communication of the occurrence of these events. We investigate the expressiveness of this new coordination mechanism and we prove that even if event notification strictly increases the expressiveness of a language with only input and output, the obtained language is still strictly less expressive than a language containing also the test for absence.

1 Introduction

In the last decades we assisted to a dramatic evolution of computing systems, leading from stand-alone mainframes to a worldwide network connecting smaller, yet much more powerful processors. The next expected step in this direction is represented by the so-called *ubiquitous computing*, based on the idea of dynamically reconfigurable federations composed of users and resources required by those users. For instance, the Jini architecture [19] represents a first proposal of Sun for a Java-based technology inspired by this new computing paradigm.

In this scenario, one of the most challenging topics is concerned with the coordination of the federated components. For this reason, a renewed interest in coordination languages – that have been around for more than fifteen years – has arisen. For example, JavaSpaces [18] and TSpaces [20] are two recent coordination middlewares for distributed Java programming proposed by Sun and IBM, respectively. These proposals incorporate the main features of both the two historical groups of coordination models [13]: the *data-driven* approach, initiated by Linda [8] and based on the notion of a shared data repository, and the *control-driven* model, advocated by Manifold [1] and centered around the concepts of raising and reaction to events. Besides the typical Linda-like

* Work partially supported by Esprit working group n.24512 "Coordina"

G. Smolka (Ed.): ESOP/ETAPS 2000, LNCS 1782, pp. 41–55, 2000.

coordination primitives (processes interact via the introduction, consumption and test for presence/absence of data inside a shared repository) both JavaSpaces and TSpaces provide event registration and notification. This mechanism allows a process to register interest in the future arrivals of a particular kind of data, and then receive communication of the occurrence of these events.

In this paper we investigate the interplay of the event notification mechanism with the classical Linda-like coordination paradigm. In particular we focus on the expressive power of event notification and we prove the existence of a hierarchy of expressiveness among the possible combinations of coordination primitives: *in*, *out*, and *inp* are strictly more expressive than *in*, *out*, and *notify*, which in turn are strictly more expressive than *in* and *out* only.

These results are proved by introducing a minimal language containing all the coordination mechanisms we are dealing with, and by considering the sublanguages corresponding to the various combinations of the coordination primitives. The complete language (denoted by $\mathbf{L}_{ntf,inp}$) is obtained by extending a Linda based process algebra presented in [2] with the event notification mechanism. We consider the following sublanguages: $\mathbf{L}$ containing only *in* and *out*, $\mathbf{L}_{ntf}$ containing also *notify*, and $\mathbf{L}_{inp}$ containing *in*, *out* and *inp*.

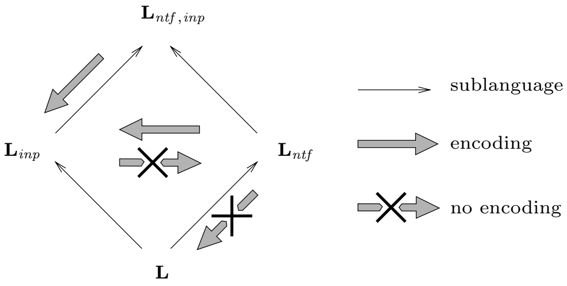


Fig. 1. Overview of the results.

The hierarchy of expressiveness sketched above follows from the three results summarized in Figure 1.

The expressiveness gap between $\mathbf{L}_{ntf}$ and $\mathbf{L}$ can be deduced by the following facts:

(1) There exists an encoding of $\mathbf{L}$ on finite Place/Transition nets [14,16] which preserves the interleaving semantics. As the existence of a terminating computation is decidable in P/T nets [6], the same holds also in $\mathbf{L}$.

(2) There exists a *nondeterministic* implementation of Random Access Machines (RAM) [17], a well known Turing powerful formalism, in $\mathbf{L}_{ntf}$. The implementation preserves the terminating behaviour: a RAM terminates if and only if the corresponding implementation has a terminating computation. Thus, the existence of a terminating computation is not decidable in $\mathbf{L}_{ntf}$.

Hence there exists no encoding of $\mathbf{L}_{ntf}$ in $\mathbf{L}$ which preserves at least the existence of a terminating computation.

The discrimination between $\mathbf{L}_{inp}$ and $\mathbf{L}_{ntf}$ proceeds in a similar way:

(3) There exists an encoding of $\mathbf{L}_{ntf}$ on finite Place/Transition nets extended with transfer arcs [7] which preserves the existence of an infinite computation. As this property is decidable in this kind of P/T nets, the same holds also in $\mathbf{L}_{ntf}$.

(4) There exists a *deterministic* implementation of RAM in $\mathbf{L}_{inp}$ such that a RAM terminates if and only if all the computation of the corresponding implementation terminate. Thus, the existence of an infinite computation is not decidable in $\mathbf{L}_{inp}$.

Hence there exists no encoding of $\mathbf{L}_{inp}$ in $\mathbf{L}_{ntf}$ which preserves at least the existence of an infinite computation.

Finally, the last result is:

(5) The event notification mechanism can be realized by means of the *inp* operator; indeed we provide an encoding of $\mathbf{L}_{ntf,inp}$ in $\mathbf{L}_{inp}$ (and hence also of $\mathbf{L}_{ntf}$ in $\mathbf{L}_{inp}$).

The paper is organized as follows. Section 2 presents the syntax and semantics of the language. Section 3, 4, and 5 discuss respectively the discriminating results between $\mathbf{L}_{ntf}$ and $\mathbf{L}$, between $\mathbf{L}_{inp}$ and $\mathbf{L}_{ntf}$, and the encoding of $\mathbf{L}_{ntf,inp}$ in $\mathbf{L}_{inp}$. Section 6 reports some conclusive remarks.

2 The Syntax and the Operational Semantics

Let *Name* be a denumerable set of message names, ranged over by $a, b, \ldots$. The syntax is defined by the following grammar:

$P ::= \langle a \rangle \ \mid \ C \ \mid \ P|P$

$C ::= \mathbf{0} \ \mid \ \mu.C \ \mid \ inp(a)?C\_C \ \mid \ C|C$

where:

$\mu ::= in(a) \ \mid \ out(a) \ \mid \ notify(a, C) \ \mid !in(a)$

Agents, ranged over by P, Q, $\ldots$, consist of the parallel composition of the data already in the dataspace (each one denoted by one agent $\langle a \rangle$) and the concurrent programs denoted by C, D, $\ldots$, that share these data. A program can be a terminated program $\mathbf{0}$ (which is usually omitted for the sake of simplicity), a prefix form $\mu.P$, an *if-then-else* form $inp(a)?P\_Q$, or the parallel composition of programs.

A prefix μ can be one of the primitives $in(a)$ or $out(a)$, indicating the withdrawing or the emission of datum a respectively, and the $notify(a, P)$ operation that registers interest in the incoming arrivals of new instances of datum a: every time a new instance of $\langle a \rangle$ is produced, a new copy of process P is spawned. We also consider the bang operator $!in(a)$ which is a form of replication guarded on input operations: the term $!in(a).P$ is always ready to consume an instance of $\langle a \rangle$ and then activate a copy of P. The if-then-else form is used to model the *inp* primitive: $inp(a)?P\_Q$ is a program which requires an instance of $\langle a \rangle$ to be

Table 1. Operational semantics (symmetric rules omitted).

$$
\begin{array}{llll}
(1) & \langle a \rangle \xrightarrow{\overline{a}} \mathbf{0} & (2) & \mathit{out}(a).P \xrightarrow{\vec{a}} \langle a \rangle | P \\[4pt]
(3) & \mathit{in}(a).P \xrightarrow{a} P & (4) & \mathit{notify}(a, Q).P \xrightarrow{\tau} \mathit{on}(a).Q | P \\[4pt]
(5) & \mathit{on}(a).P \xrightarrow{\dot{a}} P | \mathit{on}(a).P & (6) & !\mathit{in}(a).P \xrightarrow{a} P | !\mathit{in}(a).P \\[4pt]
(7) & \mathit{inp}(a)?P\_Q \xrightarrow{a} P & (8) & \mathit{inp}(a)?P\_Q \xrightarrow{\neg a} Q \\[4pt]
(9) & \dfrac{P \xrightarrow{\overline{a}} P' \quad Q \xrightarrow{a} Q'}{P|Q \xrightarrow{\tau} P'|Q'} & (10) & \dfrac{P \xrightarrow{\neg a} P' \quad Q \not\xrightarrow{\overline{a}}}{P|Q \xrightarrow{\neg a} P'|Q} \\[12pt]
(11) & \dfrac{P \xrightarrow{\dot{a}} P' \quad Q \xrightarrow{\dot{a}} Q'}{P|Q \xrightarrow{\dot{a}} P'|Q'} & (12) & \dfrac{P \xrightarrow{\dot{a}} P' \quad Q \not\xrightarrow{\dot{a}}}{P|Q \xrightarrow{\dot{a}} P'|Q} \\[12pt]
(13) & \dfrac{P \xrightarrow{\vec{a}} P' \quad Q \xrightarrow{\dot{a}} Q'}{P|Q \xrightarrow{\vec{a}} P'|Q'} & (14) & \dfrac{P \xrightarrow{\vec{a}} P' \quad Q \not\xrightarrow{\dot{a}}}{P|Q \xrightarrow{\vec{a}} P'|Q}
\end{array}
$$

$$
(15) \qquad \frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} \qquad \alpha \neq \neg a, \vec{a}, \dot{a}
$$

consumed; if it is present, the program P is executed, otherwise Q is chosen. In the following, *Agent* denotes the set containing all possible agents.

The semantics of the language is described via a labeled transition system $(\mathit{Agent}, \mathit{Label}, \longrightarrow)$ where $\mathit{Label} = \{\tau\} \cup \{a, \overline{a}, \neg a, \vec{a}, \dot{a} \;\mid\; a \in \mathit{Name}\}$ (ranged over by α, β, $\ldots$) is the set of the possible labels. The labeled transition relation $\longrightarrow$ is the smallest one satisfying the axioms and rules in Table 1. For the sake of simplicity we have omitted the symmetric rules of $(9) - (15)$.

Axiom (1) indicates that $\langle a \rangle$ is able to give its contents to the environment by performing an action labeled with $\overline{a}$. Axiom (2) describes the output: in one step a new datum is produced and the corresponding continuation is activated. The production of this new instance of $\langle a \rangle$ is communicated to the environment by decorating this action with the label $\vec{a}$. Axiom (3) associates to the action performed by the prefix $\mathit{in}(a)$ the label a, which is the complementary of $\overline{a}$.

Axiom (4) indicates that $\mathit{notify}(a, P)$ produces a new kind of agent $\mathit{on}(a).P$ (that we add to the syntax as an auxiliary term). This process spawns a new instance of P every time a new $\langle a \rangle$ is produced. This behaviour is described in axiom (5) where the label $\dot{a}$ is used to describe this kind of computation step. The term $!\mathit{in}(a).P$ is able to activate a new copy of P by performing an action labeled with a that requires an instance of $\langle a \rangle$ to be consumed (axiom (6)).

Axioms (7) and (8) describe the semantics of $\mathit{inp}(a)?P\_Q$: if the required $\langle a \rangle$ is present it can be consumed (axiom (7)), otherwise its absence is guessed by performing an action labeled with $\neg a$ (axiom (8)). Rule (9) is the usual synchronization rule.

Rules (10) – (14) regard the way actions labeled with the non-standard labels $\neg a$, $\dot{a}$, and $\vec{a}$ are inferred to structured terms. Rule (10) indicates that actions labeled with $\neg a$ can be performed only if no $\langle a \rangle$ is present in the environment (i.e. no transition labelled with $\overline{a}$ can be performed). Rules (11) and (12) consider actions labelled with $\dot{a}$ indicating the interest in the incoming instances of $\langle a \rangle$. If one process able to perform this kind of action is composed in parallel with another one registered for the same event their local actions are combined in a global one (rule (11)); otherwise, the process performs its own action leaving the environment unchanged (rule (12)). Rules (13) and (14) deal with two different cases regarding the label $\vec{a}$ indicating the arrival of a new instance of $\langle a \rangle$: if processes waiting for the notification of this event are present in the environment they are waked-up (rule (13)); otherwise, the environment is left unchanged (rule (14)). The last rule (15) is the standard local rule that can be applied only to actions different from the non-standard $\neg a$, $\vec{a}$, and $\dot{a}$.

Note that rules (10), (12), and (14) use negative premises; however, our operational semantics is well defined, because our transition system specification is strictly stratifiable [9], condition that ensures (as proved in [9]) the existence of a unique transition system agreeing with it.

We define a *structural congruence* (denoted by $\equiv$) as the minimal congruence relation satisfying the monoidal laws for the parallel composition operator:

$$P \equiv P|\mathbf{0} \qquad P|Q \equiv Q|P \qquad P|(Q|R) \equiv (P|Q)|R$$

As two structural congruent agents are observationally indistinguishable, in the remainder of the paper we will reason up to structural congruence.

In the following we will only consider computations consisting of reduction steps, i.e., the internal derivations that a *stand-alone* agent is able to perform independently of the context. In our language, we consider as reductions not only the usual derivations labeled with τ, but also the non-standard labeled with $\neg a$ and $\vec{a}$. In fact, derivation $P \xrightarrow{\neg a} P'$ indicates that P can become P' if no $\langle a \rangle$ is available in the external environment, and $P \xrightarrow{\vec{a}} P'$ describes that a new agent $\langle a \rangle$ has been produced. Hence, in any of these cases, if P is stand-alone (i.e. without external environment) it is able to become P'. Indeed, these labels have been used only for helping a SOS [15] formulation of the semantics, but they correspond conceptually to internal steps. Formally, we define reduction steps as follows:

$P \longrightarrow P'$ iff $P \xrightarrow{\tau} P'$ or $P \xrightarrow{\neg a} P'$ or $P \xrightarrow{\vec{a}} P'$ for some a

We use $P \not\longrightarrow$ to state that there exists no P' such that $P \longrightarrow P'$.

An agent P has a terminating computation (denoted by $P \downarrow$) if it can block after a finite amount of internal steps: $P \longrightarrow^* P'$ with $P' \not\longrightarrow$. On the other hand, an agent P has an infinite computation (denoted by $P \uparrow$) if there exists an infinite computation starting from P: for each natural index i there exists P_i such that $P = P_0$ and $P_i \longrightarrow P_{i+1}$. Observe that due to the nondeterminism of our languages the two above conditions are not in general mutually exclusive, i.e., given a process P both $P \downarrow$ and $P \uparrow$ may hold.

3 Comparing $\mathbf{L}_{ntf}$ and L

The discrimination between $\mathbf{L}_{ntf}$ and **L** is a direct consequence of the facts (1) and (2) listed in the Introduction.

The proof of (1) is a trivial adaptation of a result presented in [4]. Indeed, as we made in that paper, it is possible to define for **L** a Place/Transition net [14,16] semantics such that for each agent P the corresponding P/T net is finite and preserves the interleaving semantics; thus, an agent can terminate if and only if the corresponding net has a terminating computation. As this property can be decided in finite P/T nets [6], we can conclude that given a process P of **L** it is decidable if $P\downarrow$.

Result (2) uses Random Access Machines (RAM) [17] which is a Turing equivalent formalism. A RAM is composed of a finite set of registers, that can hold arbitrary large natural numbers, and by a program, that is a sequence of simple numbered instructions, like arithmetical operations (on the contents of registers) or conditional jumps.

To perform a computation, the inputs are provided in registers $r_1, \ldots, r_m$; if other registers $r_{m+1}, \ldots, r_n$ are used in the program, they are supposed to contain the value 0 at the beginning of the computation. The execution of the program begins with the first instruction and continues by executing the other instructions in sequence, unless a jump instruction is encountered. The execution stops when an instruction number higher than the length of the program is reached. If the program terminates, the result of the computation is the contents of the registers.

In [12] it is shown that the following two instructions are sufficient to model every recursive function:

- $Succ(r_j)$: adds 1 to the content of register r_j;
- $DecJump(r_j, s)$: if the content of register r_j is not zero, then decreases it by 1 and go to the next instruction, otherwise jumps to instruction s.

We present an encoding of RAM based on the *notify* primitive. The encoding we present is *nondeterministic* as it introduces some extra infinite computations; nevertheless, it is ensured that a RAM terminates if and only if the corresponding encoding has a terminating computation. As termination cannot be decided in Turing equivalent formalisms, the same holds also for $\mathbf{L}_{ntf}$. A question remains open in this section: "Is it possible to define in $\mathbf{L}_{ntf}$ a more adequate deterministic implementation of RAM which preserves also the divergent behaviour?". The answer is no, and it is motivated in Section 4 where we prove that the presence of an infinite computation can be decided in $\mathbf{L}_{ntf}$. On the other hand, we will show in the same Section that a deterministic implementation of RAM can be defined in $\mathbf{L}_{inp}$.

The encoding implements nondeterministically *DecJump* operations: two possible behaviours can be chosen, the first is valid if the tested register is not zero, the second otherwise. If the wrong choice is made, the computation is ensured to be infinite; in this case, we cannot say anything about the corresponding RAM.

Nevertheless, if the computation terminates, it is ensured that it corresponds to the computation of the corresponding RAM. Conversely, any computation of the RAM is simulated by the computation of the corresponding encoding in which no wrong choice is performed.

Table 2. Encoding RAM in $\mathbf{L}_{ntf}$.

$[\![R]\!]$	$= [\![I_1]\!] \mid \ldots \mid [\![I_k]\!] \mid in(loop).DIV$
$[\![i : Succ(r_j)]\!]$	$= !in(p_i).out(r_j).notify(zero_j, INC).out(p_{i+1})$
$[\![i : DecJump(r_j, s)]\!]$	$= !in(p_i).out(loop).in(r_j).in(loop).notify(zero_j, DEC).out(p_{i+1})$
	$\mid !in(p_i).out(zero_j).in(zero_j).out(p_s)$
where:	
INC	$= out(loop).in(match).in(loop)$
DEC	$= out(match)$
DIV	$= out(div).!in(div).out(div)$

Given the RAM program R composed by the instructions $I_1 \ldots I_k$ the corresponding encoding is defined in Table 2. Observe that DIV is an agent that cannot terminate; we will prove that it is activated whenever a wrong choice is made.

The basic idea of this encoding is to represent the actual content of each register r_j with a corresponding number of $\langle r_j \rangle$. Moreover, every time an increment (or a decrement) on the register r_j is performed, a new agent $on(zero_j).INC$ (or $on(zero_j).DEC$) is spawned by using the *notify* operation. In this way it is possible to check if the actual content of a register r_j is zero by verifying if the occurrences of $on(zero_j).INC$ corresponds to the ones of $on(zero_j).DEC$.

There are two possible wrong choices that can be performed during the computation: (i) a decrement on a register containing zero or (ii) a jump for zero on a non-empty register.

In the case (i), $out(loop).in(r_j).in(loop).notify(zero_j, DEC).out(p_{i+1})$ is activated with no $\langle r_j \rangle$ available. Thus, the program produces $\langle loop \rangle$ and blocks trying to execute $in(r_j)$. The produced $\langle loop \rangle$ will be not consumed and the agent DIV will be activated.

In the case (ii), the process $out(zero_j).in(zero_j).out(p_s)$ is activated when there are more occurrences of the auxiliary agent $on(zero_j).INC$ than the ones of $on(zero_j).DEC$. When $\langle zero_j \rangle$ is emitted, its production is notified to the auxiliary agents; then the corresponding processes INC and DEC start. Each DEC emits an agent $\langle match \rangle$ while each INC produces a term $\langle loop \rangle$, and requires a $\langle match \rangle$ to be consumed before removing the emitted $\langle loop \rangle$. As there are more INC processes than DEC, one of the processes INC will block waiting for an unavailable $\langle match \rangle$; thus it will not consume its corresponding $\langle loop \rangle$. As before, DIV will be activated.

The formal proof of correctness of the encoding requires a representation of the actual state of a RAM computation: we use $(i, inc_1, dec_1, \ldots, inc_m, dec_m)$, where i is the index of the next instruction to execute while for each register index l, inc_l (resp. dec_l) represents the number of increments (resp. decrements) that have been performed on the register r_l. The actual content of r_l corresponds to $inc_l - dec_l$. In order to deal with correct configurations only, we assume that the number of increments is greater or equal than the number of decrements.

Given a RAM program R, we write
$((i, inc_1, dec_1, \ldots, inc_n, dec_n), R) \longrightarrow ((i', inc'_1, dec'_1, \ldots, inc'_n, dec'_n), R)$
to state that the computation moves from the first to the second configuration by performing the i^{th} instruction of R; $((i, inc_1, dec_1, \ldots, inc_n, dec_n), R) \not\longrightarrow$ means that the program R has no instruction i, i.e., the computation is terminated. As RAM computations are deterministic, given a RAM program R and a configuration $(i, inc_1, dec_1, \ldots, inc_m, dec_m)$, the corresponding computation will either terminate (denoted by $((i, inc_1, dec_1, \ldots, inc_m, dec_m), R) \downarrow$) or diverge ($((i, inc_1, dec_1, \ldots, inc_m, dec_m), R) \uparrow$). As RAM permits to model all the computable functions both the termination and the divergence of a computation are not decidable.

According to this representation technique a configuration is modeled as follows:
$[\![(i, inc_1, dec_1, \ldots, inc_n, dec_n)]\!] =$
$\langle p_i \rangle | \prod_{i=1 \ldots n} (\prod_{inc_i} on(zero_i).INC | \prod_{dec_i} on(zero_i).DEC | \prod_{inc_i - dec_i} \langle r_j \rangle)$
where $\prod_{i \in I} P_i$ denotes the parallel composition of the indexed terms P_i.

It is not difficult to prove the following lemma stating that the encoding is complete as each RAM computation can be simulated by the corresponding encoding.

Theorem 1. *Let R be a RAM program, if*
$((i, inc_1, dec_1, \ldots, inc_n, dec_n), R) \longrightarrow ((i', inc'_1, dec'_1, \ldots, inc'_n, dec'_n), R)$
then also
$[\![(i, inc_1, dec_1, \ldots, inc_n, dec_n)]\!] | [\![R]\!] \longrightarrow^* [\![(i', inc'_1, dec'_1, \ldots, inc'_n, dec'_n)]\!] | [\![R]\!]$

On the other hand the encoding is not sound as it introduces infinite computations. Nevertheless, a weaker soundness for terminating computations holds.

Theorem 2. *Let R be a RAM program, if*
$[\![(i, inc_1, dec_1, \ldots, inc_n, dec_n)]\!] | [\![R]\!] \longrightarrow^* P \not\longrightarrow$
then $P = [\![(i', inc'_1, dec'_1, \ldots, inc'_n, dec'_n)]\!] | [\![R]\!]$ such that
$((i, inc_1, dec_1, \ldots, inc_n, dec_n), R) \longrightarrow^* ((i', inc'_1, dec'_1, \ldots, inc'_n, dec'_n), R) \not\longrightarrow$

Corollary 1. *Let R be a RAM program, then*
$((i, inc_1, dec_1, \ldots, inc_n, dec_n), R) \downarrow$ *iff* $[\![(i, inc_1, dec_1, \ldots, inc_n, dec_n)]\!] | [\![R]\!] \downarrow$

4 Comparing $\mathbf{L}_{inp}$ and $\mathbf{L}_{ntf}$

The discrimination between $\mathbf{L}_{inp}$ and $\mathbf{L}_{ntf}$ is a direct consequence of the facts (3) and (4) listed in the Introduction.

The result (4) has been already proved in [4]. In that paper an encoding of RAM in a language corresponding to $\mathbf{L}_{inp}$ is presented. Also that encoding (that we do not report here due to the space limits) represents the content of register r_j by means of agents of kind $\langle r_j \rangle$. In this way, a *DecJump* instruction testing the register r_j can be simply implemented by means of an $inp(r_j)$ operation which either consumes an available $\langle r_j \rangle$ or observes that the register is empty. In [4] we prove that a RAM program can perform a computation step if and only if its encoding can perform the corresponding step.

In order to prove the result (3) we recall, using a notation convenient for our purposes, the definition of simple P/T nets extended with transfer arcs (see, e.g., [7]).

Definition 1. *Given a set S, we denote by $\mathcal{M}_{fin}(S)$ the set of the finite multisets on S and by $\mathcal{F}_p(S, S)$ the set of the partial functions defined on S. We use $\oplus$ to denote multiset union. A P/T net with transfer arcs is a triple $N = (S, T, m_0)$ where S is the set of* places*, T is the set of* transitions *(which are triples $(c, p, f) \in \mathcal{M}_{fin}(S) \times \mathcal{M}_{fin}(S) \times \mathcal{F}_p(S, S)$ such that the domain of the partial function f has no intersection with c and p), and m_0 is a finite multiset of places. Finite multisets over the set S of places are called* markings*; m_0 is called* initial marking*. Given a marking m and a place s, $m(s)$ denotes the number of occurrences of s inside m and we say that the place s contains $m(s)$* tokens*. A P/T net with transfer arcs is finite if both S and T are finite.*

A transitions $t = (c, p, f)$ is usually written in the form $c \overset{f}{\rightarrowtail} p$ and f is omitted when empty. The marking c is called the preset *of t and represents the tokens to be* consumed*. The marking p is called the* postset *of t and represents the tokens to be* produced*. The partial function f denotes the* transfer arcs *of the transition which connect each place s in the domain of f to its image $f(s)$. The meaning of f is the following: when the transition fires all the tokens inside a place s in the domain of f are transferred to the connected place $f(s)$.*

A transition $t = (c, p, f)$ is enabled *at m if $c \subseteq m$. The execution of the transition produces the new marking m' such that $m'(s) = m(s) - c(s) + p(s) + \sum_{s':f(s')=s} m(s')$, if s is not in the domain of f, $m'(s) = \sum_{s':f(s')=s} m(s')$, otherwise. This is written as $m \xrightarrow{t} m'$ or simply $m \longrightarrow m'$ when the transition t is not relevant. We use σ, σ' to range over sequences of transitions; the empty sequence is denoted by ε; let $\sigma = t_1, \dots, t_n$, we write $m \xrightarrow{\sigma} m'$ to mean the* firing sequence *$m \xrightarrow{t_1} \cdots \xrightarrow{t_n} m'$. The net $N = (S, T, m_0)$ has an* infinite *computation if it has a legal infinite firing sequence.*

The basic idea underlying the definition of an operational net semantics for a process algebra is to decompose a process P into a multiset of sequential components, which can be thought of as running in parallel. Each sequential component has a corresponding place in the net, and will be represented by a token in that place. Reductions are represented by transitions which consume and produce multisets of tokens.

In our particular case we deal with different kinds of sequential components: programs of the form $\mu.P$ or $inp(a)?P\_Q$, agents $\langle a\rangle$, and terms $on(a,P)$ representing idle processes $on(a).P$. Besides these classes of components corresponding directly to terms of the language, we need to introduce a new kind of components $arrived(a,P)$ used to model event notification.

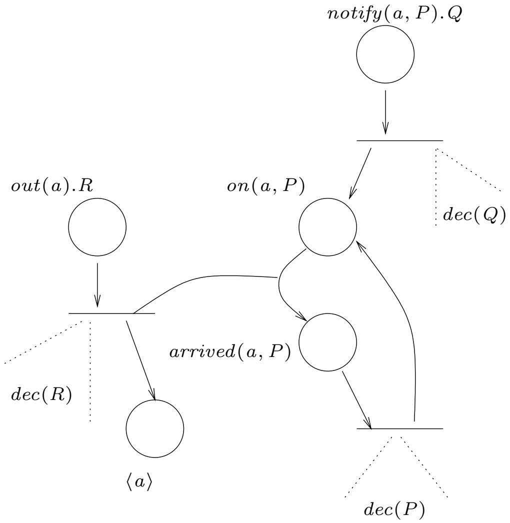


Fig. 2. Modeling event notification.

The way we represent input and output operations in our net semantics is standard. More interesting is the mechanism used to model event notification represented in Figure 2. Whenever a new token is introduced in the place $\langle a\rangle$, each token in a place $on(a,P)$ is transferred to the corresponding place $arrived(a,P)$. In order to realize this, we use a *transfer* arc that moves all the tokens inside the source place to the target one. Each token introduced in $arrived(a,P)$ will be responsible for the activation of the new instance of P. Moreover, when the activation happens, also a token in $on(a,P)$ is introduced in order to register interest in the next production of a token in $\langle a\rangle$.

The main drawback of this procedure used to model event notification is that it is not executed atomically. For instance, a new token in $\langle a\rangle$ can be produced before it terminates. In this case, the processes whose corresponding token is still in the place $arrived(a,P)$ will be not notified of the occurrence of this event. However, as we will prove in the following, even in the presence of this drawback the net semantics respects the existence of infinite computation.

After the informal description of the net semantics we introduce its formal definition. Given the agent P, we define the corresponding contextual P/T system $Net(P)$. In order to do this, we need the following notations.

- Let $\mathcal{S}$ be the set
  $\{P \mid\ P$ sequential program$\} \cup \{\langle a\rangle \mid\ a$ message name$\} \cup$
  $\{on(a,P), arrived(a,P) \mid\ a$ message name and P program$\}$.
- Let the function $dec : Agent \to \mathcal{M}_{fin}(\mathcal{S})$ be the decomposition of agents into markings, reported in Table 3.
- Let $\mathcal{T}$ contain the transitions obtained as instances of the axiom schemata presented in Table 4.

The axioms in Table 3, describing the decomposition of agents, state that the agent $\mathbf{0}$ generates no tokens; the decomposition of the terms $\langle a\rangle$ and of the other processes produces one token in the corresponding place; the decomposition of the idle process $on(a).P$ generates one token in place $on(a,P)$; and the parallel composition is interpreted as multiset union, i.e, the decomposition of $P|Q$ is $dec(P) \oplus dec(Q)$.

The axioms in Table 4 define the possible transitions. Axiom `in(a,Q)` deals with the execution of the primitives $in(a)$: a token from place $\langle a\rangle$ is consumed. Axiom `out(a,Q)` describes how the emission of new datum is obtained: a new token in the place $\langle a\rangle$ is introduced and the transfer arcs move all the tokens from the places $on(a,R)$ in the corresponding $arrived(a,R)$. In this way, all the idle agents are notified. The activation of the corresponding processes R requires a further step described by the axiom `arrived(a,Q)`: an instance of process Q is activated (by introducing tokens in the corresponding places) and a token is reintroduced in the place $on(a,\,Q)$ in order to register interest in the next token produced in $\langle a\rangle$. Axiom `!in(a,Q)` deals with the bang operator: if a token is present in place $!in(a).Q$ and a token can be consumed from place $\langle a\rangle$, then a new copy of $dec(Q)$ is produced and a token is reintroduced in $!in(a).Q$. Finally, axiom `notify(a,Q,R)` produces a token in the place $on(a,\,Q)$ in order to register interest in the arrival of the future incoming token in $\langle a\rangle$.

Definition 2. *Let P be an agent. We define the triple $Net(P) = (S, T, m_0)$ where:*

$$\begin{aligned}
S\ &= \{Q \mid Q \textit{ sequential subprogram of } P\} \cup \\
&\quad\ \{\langle a\rangle \mid a \textit{ message name in } P\} \cup \\
&\quad\ \{on(a,\,Q), arrived(a,\,Q) \mid a \textit{ message name in } P \textit{ and } Q \textit{ subprogam of } P\} \\
T\ &= \{c \overset{f|_S}{\rightarrowtail} p \mid c \overset{f}{\rightarrowtail} p \in \mathcal{T} \textit{ and } dom(c) \subseteq S\} \\
m_0 &= dec(P)
\end{aligned}$$

where by $f|_S$ we mean the restriction of function f to its subdomain S.

It is not difficult to see that $Net(P)$ is well defined, in the sense it is a correct P/T net with transfer arcs; moreover, it is finite. Moreover the net semantics is complete as it simulates all the possible computations allowed by the operational semantics.

Table 3. Decomposition function.

$dec(\mathbf{0})$	$= \emptyset$	$dec(\langle a \rangle)$	$= \{\langle a \rangle\}$
$dec(\mu.P)$	$= \{\mu.P\}$	$dec(on(a).P)$	$= \{on(a, P)\}$
$dec(P\|Q)$	$= dec(P) \oplus dec(Q)$		

Table 4. Transition specification.

`in(a,Q)`	$in(a).Q \oplus \langle a \rangle \rightarrowtail dec(Q)$
`out(a,Q)`	$out(a).Q \overset{f}{\rightarrowtail} \langle\langle a \rangle\rangle \oplus dec(Q)$
	where $f = \{(on(a, R), arrived(a, R)) \mid R$ is a program$\}$
`arrived(a,Q)`	$arrived(a, Q) \rightarrowtail dec(Q) \oplus on(a, Q)$
`!in(a,Q)`	$!in(a).Q \oplus \langle a \rangle \rightarrowtail\, !in(a).Q \oplus dec(Q)$
`notify(a,Q,R)`	$notify(a, Q).R \rightarrowtail on(a, Q) \oplus dec(R)$

Theorem 3. *Let* $Net(P) = (S, T, m_0)$ *and* R *be an agent s.t.* $dom(dec(R)) \subseteq S$. *If* $R \longrightarrow R'$ *then there exists a transition sequence* σ *s.t.* $dec(R) \xrightarrow{\sigma} dec(R')$.

The above theorem proves the completeness of the net semantics which, on the other hand, is not sound. Indeed, as we have already discussed, the encoding introduces some slightly different computations due to the non atomicity of the way we model the event notification mechanism. However, the introduction of these computations does not alterate the possibility to have an infinite computation. This is proved by the following Theorem.

Theorem 4. *Let* $Net(P) = (S, T, m_0)$ *and* R *an agent s.t.* $dom(dec(R)) \subseteq S$. *There exists an infinite firing sequence starting from* $dec(R)$ *iff* $R \uparrow$.

5 Comparing $\mathbf{L}_{ntf,inp}$ and $\mathbf{L}_{inp}$

In Section 3 we proved that *in* and *out* are not sufficiently powerful to encode the event notification mechanism; now we show that the addition of the *inp* operation permits to realize the encoding of $\mathbf{L}_{ntf,inp}$ in $\mathbf{L}_{inp}$.

In order to simulate event notification we force each process performing a $notify(a, P)$ to declare its interest in the incoming $\langle a \rangle$ by emitting $\langle wait_a \rangle$. Then, the process remains idle, waiting for $\langle arrived_a \rangle$, signaling that an instance of $\langle a \rangle$ appeared. When an output operation $out(a)$ is performed, a protocol composed of three phases is started.

Table 5. Encoding the *notify* primitive ($n(P)$ denotes the set of message names of P).

$$[\![P]\!] = [\![P]\!] | ME_{n(P)}$$

$$[\![\mathbf{0}]\!] = \mathbf{0} \qquad [\![out(a).P]\!] = in(me_a).out(wc_{aP}) | O(a,P)$$

$$[\![\langle a \rangle]\!] = \langle a \rangle \qquad [\![inp(a)?P\_Q]\!] = inp(a)?[\![P]\!]\_[\![Q]\!]$$

$$[\![in(a).P]\!] = in(a).[\![P]\!] \qquad [\![on(a).P]\!] = \langle wait_a \rangle | W(a,P) | !in(w_{aP}).W(a,P)$$

$$[\![!in(a).P]\!] = !in(a).[\![P]\!] \qquad [\![P|Q]\!] = [\![P]\!] | [\![Q]\!]$$

$$[\![notify(a,P).Q]\!] = in(me_a).out(wait_a).out(w_{aP}).out(me_a).(!in(w_{aP}).W(a,P) | [\![Q]\!])$$

$$ME_A = \textstyle\prod_{a \in A} \langle me_a \rangle$$

$$W(a,P) = in(arrived_a).out(wait_a).out(ack_a).out(w_{aP}).[\![P]\!]$$

$$\begin{aligned} O(a,P) = {} & !in(wc_{aP}).inp(wait_a)?(out(creating_a).out(wc_{aP}))\_(out(a).out(ca_{aP}))| \\ & !in(ca_{aP}).inp(creating_a)?(out(arrived_a).out(askack_a).out(ca_{aP}))\_\,out(ea_{aP})| \\ & !in(ea_{aP}).inp(askack_a)?(in(ack_a).out(ea_{aP}))\_(out(me_a).[\![P]\!]) \end{aligned}$$

In the first phase, each $\langle wait_a \rangle$ is replaced by $\langle creating_a \rangle$. At the end of this phase $\langle a \rangle$ is produced.

In the second phase, we start transforming each $\langle creating_a \rangle$ in the pair of agents $\langle arrived_a \rangle$ and $\langle askack_a \rangle$.

The agents $\langle arrived_a \rangle$ will wake up the processes that were waiting for the notification of the addition of $\langle a \rangle$; each of these processes produces a new instance of $\langle wait_a \rangle$ (to be notified of the next emissions of $\langle a \rangle$) and an $\langle ack_a \rangle$, to inform that it has been waked. We use two separated renaming phases (from $wait_a$ to $creating_a$ and then to $arrived_a$) in order to avoid that a just waked process (that has emitted $\langle wait_a \rangle$ to be notified of the next occurrence of output of a) is waked two times.

In the third phase the $\langle ack_a \rangle$ emitted by the waked processes are matched with the $\langle askack_a \rangle$ emitted in the second phase; this ensures that all the processes waiting for emission of $\langle a \rangle$ have been waked.

The concurrent execution of two or more output protocols could provoke undesired behaviour (for example, it may happen that some waiting process is notified of a single occurrence of output, instead of two); for this reason the output protocol is performed in mutual exclusion with other output protocols producing a datum with the same name. For similar reasons we avoid also the concurrent execution of the output protocol with a notification protocol concerning the same kind of datum. This is achieved by means of $\langle me_a \rangle$, which is consumed at the beginning of the protocol and reproduced at the end.

Note that, in the implementation of this protocol, the *inp* operator is necessary in order to apply a transformation to all the occurrences of a datum in the dataspace. Indeed, with only a blocking input *in* it is not possible to solve this problem. The formal definition of the encoding is presented in Table 5.

The proof of the correctness of the encoding is essentially based on an intermediate mapping, where partially executed out and notify protocols are represented with an abstract notation. We report here only the enunciates of the main results.

The following theorem states that each move performed by a process in $\mathbf{L}_{ntf,inp}$ can be mimicked by a sequence of moves of its encoding.

Theorem 5. *Let P be a term of $\mathbf{L}_{ntf,inp}$ s.t. $n(P) \subseteq A$. If $P \longrightarrow P'$ then* $[\![P]\!]|ME_A|\prod_{i=1\dots k} O(a_i, P_i) \longrightarrow^+ [\![P']\!]|ME_A|\prod_{i=1\dots h} O(b_i, Q_i)$.

The next result says that any computation of the encoding of P can be extended in order to reach the encoding of a process reachable from P.

Theorem 6. *Let P be a term of $\mathbf{L}_{ntf,inp}$ s.t. $n(P) \subseteq A$. If $[\![P]\!]|ME_A|\prod_{i=1\dots k} O(a_i, P_i) \longrightarrow^* Q$ then there exists P' such that $P \longrightarrow^* P'$ and $Q \longrightarrow^* [\![P']\!]|ME_A|\prod_{i=1\dots h} O(b_i, Q_i)$.*

6 Conclusion

We investigated the expressiveness of event notification in a data-driven coordination model. We proved that the addition of the *notify* primitive strictly increases the expressiveness of a language with only *in* and *out*, but leaves it unchanged if the language contains also *inp*. On the other hand, we showed that the *inp* primitive cannot be encoded by *in*, *out*, and *notify*.

We embedded the coordination primitives in a minimal language. The relevance of our results extends to richer languages in the following way. The encodability result extends to any language comprising the minimal features of our calculus. The negative results of non-encodability can be interpreted on a Turing complete language as the necessity for an encoding to exploit the specific computational features of the considered language.

We think that this kind of results has not only a theoretical relevance, but they could be of interest also for designers and implementors of coordination languages. For example, the powerful *inp* primitive has been a source of problems during the first distributed implementations of Linda (see, e.g., [10]). The results proved here suggest that the *notify* primitive may represent a good compromise between easiness of implementation and expressive power.

In [3] we consider three different interpretations for the *out* operation and in [4] we found an expressiveness gap between two of them. More precisely, we proved that a language with *in*, *out*, and *inp* is Turing powerful under the *ordered* semantics (the one considered here), while it is not under the *unordered* one (where the emission and the effective introduction of data in the dataspace are two independent steps). In [5] we investigate the impact of event notification on the unordered semantics: we prove that the addition of the *notify* primitive makes the language Turing powerful also under the *unordered* interpretation and it permits a faithful encoding of the ordered semantics on top of the unordered one.

Here, we have chosen the ordered interpretation as it is the semantics adopted by the actual JavaSpaces specifications, as indicated in the sections 2.3 and 2.8 of [18], and also confirmed us by personal communications with John McClain of Sun Microsystems Inc. [11].

References

1. F. Arbab, I. Herman, and P. Spilling. An overview of Manifold and its implementation. *Concurrency: Practice and Experience*, 5(1):23–70, 1993.
2. N. Busi, R. Gorrieri, and G. Zavattaro. A Process Algebraic View of Linda Coordination Primitives. *Theoretical Computer Science*, 192(2):167–199, 1998.
3. N. Busi, R. Gorrieri, and G. Zavattaro. Comparing Three Semantics for Linda-like Languages. *Theoretical Computer Science*, to appear. An extended abstract appeared in *Proc. of Coordination'97*.
4. N. Busi, R. Gorrieri, and G. Zavattaro. On the Expressiveness of Linda Coordination Primitives. *Information and Computation*, to appear. An extended abstract appeared in *Proc. of Express'97*.
5. N. Busi and G. Zavattaro. Event Notification in Data-driven Coordination Languages: Comparing the Ordered and Unordered Interpretation. In *Proc. of SAC2000*, ACM press. To appear.
6. A. Cheng, J. Esparza, and J. Palsberg. Complexity results for 1-safe nets. *Theoretical Computer Science*, 147:117–136, 1995.
7. C. Dufourd, A. Finkel, and P. Schnoebelen. Reset nets between decidability and undecidability. In *Proc. of ICALP'98*, volume 1061 of *Lecture Notes in Computer Science*, pages 103–115. Springer-Verlag, Berlin, 1998.
8. D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
9. J.F. Groote. Transition system specifications with negative premises. *Theoretical Computer Science*, 118:263–299, 1993.
10. J. Leichter. *Shared Tuple Memories, Shared Memories, Buses and LANS: Linda Implementations Across the Spectrum of Connectivity.* PhD thesis, Yale University Department of Computer Science, 1989.
11. J. McClain. Personal communications. March 1999.
12. M.L. Minsky. *Computation: finite and infinite machines.* Prentice-Hall, 1967.
13. G.A. Papadopoulos and F. Arbab. Coordination Models and Languages. *Advances in Computers*, 46:329–400, 1998.
14. C. A. Petri. *Kommunikation mit Automaten.* PhD thesis, Institut für Instrumentelle Mathematik, Bonn, Germany, 1962.
15. G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.
16. W. Reisig. *Petri Nets: An Introduction.* EATCS Monographs in Computer Science. Springer-Verlag, Berlin, 1985.
17. J.C. Shepherdson and J.E. Sturgis. Computability of recursive functions. *Journal of the ACM*, 10:217–255, 1963.
18. Sun Microsystem, Inc. *JavaSpaces Specifications*, 1998.
19. Sun Microsystem, Inc. *Jini Architecture Specifications*, 1998.
20. P. Wyckoff, S.W. McLaughry, T.J. Lehman, and D.A. Ford. T Spaces. *IBM Systems Journal*, 37(3), 1998.

Flow-Directed Closure Conversion for Typed Languages

Henry Cejtin, Suresh Jagannathan, and Stephen Weeks

[1] Entertainment Decisions, Inc. henry@clairv.com
[2] NEC Research Institute, suresh@research.nj.nec.com
[3] Intertrust STAR Laboratories, sweeks@intertrust.com

Abstract. This paper presents a new closure conversion algorithm for simply-typed languages. We have have implemented the algorithm as part of MLton, a whole-program compiler for Standard ML (SML). MLton first applies all functors and eliminates polymorphism by code duplication to produce a simply-typed program. MLton then performs closure conversion to produce a first-order, simply-typed program. In contrast to typical functional language implementations, MLton performs most optimizations on the first-order language, *after* closure conversion. There are two notable contributions of our work:

1. The translation uses a general flow-analysis framework which includes OCFA. The types in the target language fully capture the results of the analysis. MLton uses the analysis to insert coercions to translate between different representations of a closure to preserve type correctness of the target language program.
2. The translation is practical. Experimental results over a range of benchmarks including large real-world programs such as the compiler itself and the ML-Kit [25] indicate that the compile-time cost of flow analysis and closure conversion is extremely small, and that the dispatches and coercions inserted by the algorithm are dynamically infrequent.

1 Introduction

This paper presents a new closure conversion algorithm for simply-typed languages. We have implemented the algorithm as part of MLton[1] , a whole-program compiler for Standard ML (SML). MLton first applies all functors and eliminates polymorphism by code duplication to produce a simply-typed program. MLton then performs closure conversion to produce a first-order, simply-typed program. Unlike typical functional language implementations, MLton performs most optimizations on the first-order language, after closure conversion. The most important benefit of this approach is that numerous optimization techniques developed for other first-order languages can be immediately applied. In addition, a simply-typed intermediate language simplifies the overall structure

[1] MLton is available under GPL from http://www.neci.nj.nec.com/PLS/MLton/.

G. Smolka (Ed.): ESOP/ETAPS 2000, LNCS 1782, pp. 56–71, 2000.

of the compiler. Our experience with MLton indicates that simply-typed intermediate languages are sufficiently expressive to efficiently compile higher-order languages like Standard ML.

An immediate question that arises in pursuing this strategy concerns the representation of closures. Closure conversion transforms a higher-order program into a first-order one by representing each procedure with a tag identifying the code to be executed (typically a code pointer) when the procedure is applied, and an environment containing the values of the procedure's free variables. The code portion of a procedure is translated to take its environment as an extra argument.

Like previous work on defunctionalization [19,3], the translation implements closures as elements of a datatype, and dispatches at call-sites to the appropriate code. We differ in that the datatypes in the target language express all procedures that may be called at the same call-site as determined by flow analysis. Consequently, the size of dispatches at calls is inversely related to the precision of the analysis.

Using dispatches instead of code pointers to express function calls has two important benefits: (1) the target language can remain simply-typed without the need to introduce existential types [16], and (2) optimizations can use different calling conventions for different procedures applied at the same call-site. However, if the simplicity and optimization opportunities afforded by using dispatches are masked by the overhead of the dispatch itself, this strategy would be inferior to one in which the code pointer is directly embedded within the closure record. We show that the cost of dispatches for the benchmarks we have measured is a small fraction of the benchmark's overall execution time. We elaborate on these issues in Sections 4 and 6.

Our approach extends the range of expressible flow analyses beyond that of previous work [26] by inserting coercions in the target program that preserve a closure's meaning, but change its type. Using coercions, the translation expresses higher-order flow information in the first-order target language in a form verifiable by the type system. Since the results of flow analysis are completely expressed in the types of the target program, ordinary optimizations performed on the target automatically take advantage of flow information computed on the source. In Section 4, we show that representations can be chosen so that coercions have no runtime cost.

Experimental results over a range of benchmarks including the compiler itself (approximately 47K lines of SML code) and the ML Kit (approximately 75K lines) indicate that the compile-time cost of flow analysis and closure conversion is small, and that local optimizations can eliminate almost all inserted coercions. Also, MLton often produces code significantly faster than the code produced by Standard ML of New Jersey [1].

The remainder of the paper is structured as follows. Section 2 describes the source and target languages for the closure converter. Section 3 defines the class of flow analyses that the translation can use. Section 4 presents the closure conversion algorithm. A detailed example illustrating the algorithm is given in Section 5. Section 6 describes MLton and presents experimental results. Sections 7 presents related work and conclusions.

2 Source and Target Languages

We illustrate our flow-directed closure conversion translation using the source language shown on the left-hand side of Figure 1. A program consists of a collection of datatype declarations followed by an expression. As in ML, a datatype declaration defines a new sum type along with constructors to create and discriminate among values of that type. The source language is a lambda calculus core augmented by constructor application, case, tuple construction, selection of tuple components, and exceptions. Exceptions are treated as elements of a datatype. The source language is simply-typed, where types are either type constructors, arrow types, or tuple types. We omit the type rules and assume that every expression and variable is annotated with its type. We write $e : \tau$ to mean that e has type τ. We write $x : \tau$ to mean that variable x has type τ. We assume that all bound variables in a program are distinct. We use *Exp*, *Bind*, *Lam*, *App*, and *Tuple* to name the sets of specific occurrences of subterms of the forms e, b, $\texttt{fn}\ x\ \texttt{=>}\ e$, $y\ z$, and $\texttt{(}\ldots\texttt{,}\ x\texttt{,}\ \ldots\texttt{)}$, respectively, in the given program (occurrences can be defined formally using paths or unique expression labels). *TyCon* names the set of datatypes declared in a program.

Like the source language, the target language (see right-hand side of Figure 1) is simply-typed, but without arrow types, since the target language does not contain lambda expressions. A target language program is prefixed by a collection of mutually recursive first-order functions, and function application explicitly specifies the first-order function to be called.

Source Language

$$\begin{array}{lcl}
C & \in & \mathit{Con} \\
t & \in & \mathit{Tycon} \\
w,x,y,z & \in & \mathit{Var} \\
\tau & ::= & t \\
 & \mid & \tau\ \texttt{->}\ \tau \\
 & \mid & \ldots\ \texttt{*}\ \tau\ \texttt{*}\ \ldots \\
P & ::= & \texttt{let}\ \ldots\ \mathit{data}\ \ldots\ \texttt{in}\ e\ \texttt{end} \\
\mathit{data} & ::= & \texttt{datatype}\ t\ \texttt{=}\ \ldots\ \texttt{|}\ C\ \texttt{of}\ \tau\texttt{|}\ \ldots \\
e & ::= & x \\
 & \mid & \texttt{let}\ x\ \texttt{=}\ b\ \texttt{in}\ e\ \texttt{end} \\
b & ::= & e \\
 & \mid & \texttt{fn}\ w\ \texttt{=>}\ e \\
 & \mid & y\ z \\
 & \mid & C\ y \\
 & \mid & \texttt{case}\ y\ \texttt{of}\ \ldots\ \texttt{|}\ C\ z\ \texttt{=>}\ e\ \texttt{|}\ \ldots \\
 & \mid & \texttt{(}\ldots\texttt{,}\ y\texttt{,}\ \ldots\texttt{)} \\
 & \mid & \texttt{\#}i\ y \\
 & \mid & \texttt{raise}\ y \\
 & \mid & e_1\ \texttt{handle}\ y\ \texttt{=>}\ e_2
\end{array}$$

Target Language

$$\begin{array}{rcl}
f & \in & \mathit{Func} \\
\tau & ::= & t \\
 & \mid & \ldots\ \texttt{*}\ \tau\ \texttt{*}\ \ldots \\
P & ::= & \texttt{let}\ \ldots\ \mathit{data}\ \ldots\ \texttt{in} \\
 & & \texttt{let}\ \ldots\ \mathit{fun}\ \ldots\ \texttt{in}\ e\ \texttt{end} \\
 & & \texttt{end} \\
\mathit{data} & ::= & \texttt{datatype}\ t\ \texttt{=}\ \ldots\ \texttt{|}\ C\ \texttt{of}\ \tau\texttt{|}\ \ldots \\
\mathit{fun} & ::= & \texttt{fun}\ f\texttt{(}\ldots\texttt{,}\ x\texttt{,}\ \ldots\texttt{)}\ \texttt{=}\ e \\
e & ::= & x \\
 & \mid & \texttt{let}\ x\ \texttt{=}\ b\ \texttt{in}\ e\ \texttt{end} \\
b & ::= & e \\
 & \mid & f\texttt{(}\ldots\texttt{,}\ y\texttt{,}\ \ldots\texttt{)} \\
 & \mid & C\ y \\
 & \mid & \texttt{case}\ y\ \texttt{of}\ \ldots\ \texttt{|}\ C\ z\ \texttt{=>}\ e\ \texttt{|}\ \ldots \\
 & \mid & \texttt{(}\ldots\texttt{,}\ y\texttt{,}\ \ldots\texttt{)} \\
 & \mid & \texttt{\#}i\ y \\
 & \mid & \texttt{raise}\ y \\
 & \mid & e_1\ \texttt{handle}\ y\ \texttt{=>}\ e_2
\end{array}$$

Fig. 1. Source and target languages.

$$\begin{array}{lcl} v \in \mathit{Value} &=& (\mathit{Lam} \times \mathit{Env}) + \mathit{Value}^* + (\mathit{Con} \times \mathit{Value}) \\ \rho \in \mathit{Env} &=& \mathit{Var} \to \mathit{Value} \end{array}$$

$$\boxed{\rho, e \hookrightarrow v/p}$$

$$\frac{}{\rho, x \hookrightarrow \rho(x)}$$

$$\frac{\rho, b \hookrightarrow v_b \qquad \rho[x \mapsto v_b], e \hookrightarrow v/p}{\rho, \texttt{let } x \texttt{ = } b \texttt{ in } e \texttt{ end} \hookrightarrow v/p}$$

$$\frac{\rho, b \hookrightarrow p}{\rho, \texttt{let } x \texttt{ = } b \texttt{ in } e \texttt{ end} \hookrightarrow p}$$

$$\boxed{\rho, b \hookrightarrow v/p}$$

$$\frac{}{\rho, \texttt{fn } w \texttt{ => } e \hookrightarrow \langle \texttt{fn } w \texttt{ => } e, \rho|_{FV(\texttt{fn } w \texttt{ => } e)} \rangle}$$

$$\frac{\rho(y) = \langle \texttt{fn } w \texttt{ => } e, \rho' \rangle \qquad \rho'[w \mapsto \rho(z)], e \hookrightarrow v}{\rho, y\ z \hookrightarrow v}$$

$$\frac{}{\rho, C\ y \hookrightarrow \langle C, (\rho(y)) \rangle}$$

$$\frac{\rho(y) = \langle C, v \rangle \qquad \rho[z \mapsto v], e \hookrightarrow v'}{\rho, \texttt{case } y \texttt{ of } \ldots \texttt{ | } C\ z \texttt{ => } e \texttt{ | } \ldots \hookrightarrow v'}$$

$$\frac{\rho(y) = \langle C, v \rangle \qquad \rho[z \mapsto v], e \hookrightarrow p}{\rho, \texttt{case } y \texttt{ of } \ldots \texttt{ | } C\ z \texttt{ => } e \texttt{ | } \ldots \hookrightarrow p}$$

$$\frac{}{\rho, \texttt{(}\ldots\texttt{, } y\texttt{, } \ldots\texttt{)} \hookrightarrow (\ldots\texttt{, } \rho(y)\texttt{, } \ldots)}$$

$$\frac{\rho(y) = (\ldots\texttt{, } v_i\texttt{, } \ldots)}{\rho, \texttt{\#}i\ y \hookrightarrow v_i}$$

$$\frac{}{\rho, \texttt{raise } y \hookrightarrow [\rho(y)]}$$

$$\frac{\rho, e_1 \hookrightarrow v}{\rho, e_1 \texttt{ handle } y \texttt{ => } e_2 \hookrightarrow v}$$

$$\frac{\rho, e_1 \hookrightarrow [v_1] \qquad \rho[y \mapsto v_1], e_2 \hookrightarrow v_2}{\rho, e_1 \texttt{ handle } y \texttt{ => } e_2 \hookrightarrow v_2}$$

Fig. 2. Source language semantics.

We specify the source language semantics via the inductively defined relations in Figure 2. For example, expression evaluation defined via the relation written $\rho, e \hookrightarrow v/p$, is pronounced "in environment ρ, expression e evaluates either to value v or an exception packet p." In this regard, the semantics of exceptions is similar to the presentation given in [15]. We write $[v]$ to denote an exception packet containing the value v. A value is either a closure, a tuple of values, or a value built by application of a datatype constructor. The semantic rules for the target language are identical except for the rule for function application:

$$\frac{[\ldots\ x_i \mapsto \rho(y_i)\ \ldots], e \hookrightarrow v/p}{\rho, f\texttt{(}\ldots\texttt{, } y_i\texttt{, } \ldots\texttt{)} \hookrightarrow v/p}$$

where $\texttt{fun } f\texttt{(}\ldots\texttt{, } x_i\texttt{, } \ldots\texttt{) = } e$ is a function declaration in the program.

3 Flow Analysis

Our flow analysis is a standard monovariant analysis that uses abstract values to approximate sets of exact values:

$$a \in \mathit{AVal} = \mathit{TyCon} + \mathcal{P}(\mathit{Lam}) + \mathit{AVal}^*$$

An abstract value may either be a *Tycon*, which represents all constructed values of that type, a set of λ-occurrences, which represents a set of closures, or a sequence of abstract values, which represents a set of tuples.

Definition 1. *An abstract value a is consistent with a type τ if and only if one of the following holds:*

1. *$a = t$ and $\tau = t$.*
2. *$a \in \mathcal{P}(\mathit{Lam})$, $\tau = \tau_1$ `->` τ_2, and for all $f \in a$, $f : \tau_1$ `->` τ_2.*
3. *$a = (\ldots,\ a_i,\ \ldots)$, $\tau = \ldots$ `*` τ_i `*` $\ldots$, a_i is consistent with τ_i for all i.*

We define our flow analysis as a type-respecting [12] function from variables, constructors, and exception packets to abstract values in the program.

Definition 2. *A flow is a function $F : (\mathit{Var} + \mathit{Con} + \{\mathsf{packet}\}) \to \mathit{AVal}$ such that*

1. *For all x in P, if $x : \tau$ then $F(x)$ is consistent with τ.*
2. *For all C in P, if C carries values of type τ then $F(C)$ is consistent with τ.*

Informally, $F(x)$ conservatively approximates the set of values that x may take on at runtime. Similarly, $F(C)$ over-approximates the set of values to which C may be applied at runtime. The special token packet models exception values; all exception values are collected into the abstract value $F(\mathsf{packet})$.

To formally specify the meaning of an analysis, we define a pair of relations by mutual induction. The first, between environments and flows ($\rho \sqsubseteq F$), describes when an environment is approximated by the flow.

$$\rho \sqsubseteq F \text{ if for all } x \in \mathit{dom}(\rho),\ \rho(x) \sqsubseteq_F F(x)$$

The second relation, between values and abstract values ($v \sqsubseteq_F a$), describes when a value is approximated by an abstract value (relative to a flow).

1. $\mathsf{C}\ v \sqsubseteq_F t$ if C is a constructor associated with datatype t, and $v \sqsubseteq_F F(\mathsf{C})$.
2. $(\ldots,\ v_i,\ \ldots) \sqsubseteq_F (\ldots,\ a_i,\ \ldots)$ if $v_i \sqsubseteq_F a_i$ for all i.
3. $\langle$`fn` x `=>` $e, \rho\rangle \sqsubseteq_F a$ if `fn` x `=>` $e \in a$ and $\rho \sqsubseteq F$.

Figure 3 defines a collection of safety constraints such that any flow meeting them will conservatively approximate the runtime behavior of the program. We use the following partial order on abstract values:

Definition 3. *$a \geq a'$ if and only if*

- *$a = t = a'$ for some $t \in \mathit{TyCon}$,*
- *$a \supseteq a'$, where $a, a' \in \mathcal{P}(\mathit{Lam})$, or*
- *$a = (\ldots,\ a_i,\ \ldots)$, $a' = (\ldots,\ a'_i,\ \ldots)$ and $a_i \geq a'_i$ for all i.*

Theorem 1. *If F is safe and $\rho \sqsubseteq F$ then*

- *if $\rho, e \hookrightarrow v$ then $v \sqsubseteq_F F(\mathit{last}(e))$.*
- *if $\rho, e \hookrightarrow [v]$ then $v \sqsubseteq_F F(\mathsf{packet})$.*
- *if $\rho, b \hookrightarrow v$ and x = $b \in P$ then $v \sqsubseteq_F F(x)$.*
- *if $\rho, b \hookrightarrow [v]$ then $v \sqsubseteq_F F(\mathsf{packet})$.*

Proof. By induction on $\rho, e \hookrightarrow v$ and $\rho, b \hookrightarrow v$ □

Definition 4. *The* last *variable of an expression, which yields the expression's value,*
is defined as follows:

$$\begin{aligned} \mathit{last}(x) &= x \\ \mathit{last}(\texttt{let } x \texttt{ = } b \texttt{ in } e \texttt{ end}) &= \mathit{last}(e) \end{aligned}$$

Definition 5. *A flow F is* safe *if and only if, for all $x = b$ in P,*

1. *if b is e, then $F(x) = F(\mathit{last}(e))$.*
2. *if b is $\texttt{fn } y \texttt{ => } e$, then $F(x) \geq \{\texttt{fn } y \texttt{ => } e\}$.*
3. *if b is $y\ z$, then for all $\texttt{fn } w \texttt{ => } e \in F(y)$,*
   a) *$F(w) \geq F(z)$, and*
   b) *$F(x) \geq F(\mathit{last}(e))$*
4. *if b is $C\ y$, then $F(C) \geq F(y)$.*
5. *if b is $x = \texttt{case } y \texttt{ of } \ldots \texttt{ | } C_i\ z_i \texttt{ => } e_i \texttt{ | } \ldots$, then for all i,*
   a) *$F(z_i) = F(C_i)$, and*
   b) *$F(x) \geq F(\mathit{last}(e_i))$*
6. *if b is $\texttt{(}\ldots\texttt{,}\ y_i\texttt{,}\ \ldots\texttt{)}$, then $F(x) = (\ldots,\ F(y_i),\ \ldots)$.*
7. *if b is $\texttt{\#}i\ y$ and $F(y) = (\ldots,\ a_i,\ \ldots)$ then $F(x) = a_i$.*
8. *if b is $\texttt{raise } y$ then $F(\mathsf{packet}) \geq Fy$.*
9. *if b is $e_1 \texttt{ handle } z \texttt{ => } e_2$ then $F(z) \geq F(\mathsf{packet})$, $F(x) \geq F(\mathit{last}(e_1))$, and $F(x) \geq F(\mathit{last}(e_2))$.*

Fig. 3. Safety constraints on flows.

The constraints are standard for a monovariant control-flow analysis [9,17] with the following two exceptions. First, rule 4 merges all arguments to a constructor. This is to avoid introducing recursive coercions, and to reduce the number of coercions performed at runtime. Second, we use "=" instead of "≥" in some flow constraints to simplify the specification of the translation, although it is straightforward to incorporate the extra generality in practice. One can also prove that for any program, there is a minimum safe flow; this corresponds to the usual 0CFA. Another example of a safe flow is the unification-based flow analysis described by Henglein [11] and used by Tolmach and Oliva [26]. We can view this analysis as adhering to the safety constraints in Figure 3 with containment (≥) replaced by equality in the rules.

4 Closure Conversion

Given a safe flow F for the following source program:

```
let ...(datatype t = ... | C of τ| ...)... in e end
```

the closure conversion algorithm produces the following target program:

```
let datatype t = ... | C of 𝒯(F(C))| ...
      ...
      datatype 𝒯(L) = ... | 𝒞(L, fn x => e) of (... * 𝒯(F(y_i)) * ...)| ...
      ...
in let ...
          fun 𝒩(fn x => e)(r, x) = let ... y_i = #i r ... in ⟦e⟧ end
          ...
    in ⟦e⟧
    end
end
```

The translation inserts one datatype declaration for each set L that appears in the range of F, with one constructor for each λ-expression in L. We write $\mathcal{T}(L)$ to denote the new datatype for L and $\mathcal{C}(L,$ `fn` x `=>` $e)$ to denote the name of the constructor corresponding to `fn` x `=>` $e \in L$. The constructor's argument has the type of the tuple of free variables of `fn` x `=>` e, that is $(\ldots, y_i, \ldots)$. We extend $\mathcal{T}$ to abstract values by defining $\mathcal{T}(t) = t$ and $\mathcal{T}((\ldots,\ a_i,\ \ldots)) = \ldots$ `*` $\mathcal{T}(a_i)$ `*` $\ldots$

The translation also creates one function declaration for each λ-expression that occurs in the source program. The name of the target language first-order function for `fn` x `=>` e is denoted by $\mathcal{N}($`fn` x `=>` $e)$. Each function extracts all the free variables of the closure record passed as the first argument, and then continues with the translated body.

The translation uses auxiliary functions $[\![\bullet]\!] : \mathit{Exp} \rightarrow\ \mathit{Exp}$ and $[\![\bullet]\!]_x : \mathit{Bind} \rightarrow \mathit{Bind}$, which appear in Figure 4. The interesting cases in the translation are for λ-expressions and application. Rule 2b builds a closure record by applying the appropriate constructor to the tuple of the procedure's free variables. Rule 2c translates an application to a dispatch on the closure record of the procedure being applied. Because the safety constraints only require containment instead of equality, the translation inserts coercions at program points where the flow becomes less precise.

The coercion function $\mathcal{X}$, defined in Figure 4, changes the representation of a value from a more precise to a less precise type. For example, the translation of an application may require coercions at two points. First, if the abstract value of the argument is more precise than the formal, a coercion is inserted to change the argument's type to the formal's. Second, a coercion is required if the abstract value of the result is more precise than the abstract value of variable to which it becomes bound.

```
1. a) $[\![$let $x$ = $b$ in $e$ end$]\!]$ = let $x$ = $[\![b]\!]_x$ in $[\![e]\!]$ end
   b) $[\![x]\!] = x$
2. a) $[\![e]\!]_x = [\![e]\!]$
   b) $[\![$fn $w$ => $e]\!]_x = C$( ... , $y$, ... ),
      where $C = \mathcal{C}(F(x),$ fn $w$ => $e)$ and $\mathsf{FV}($fn $w$ => $e) = \ldots\ y\ \ldots$.
   c) $[\![y\ z]\!]_x$ = case $y$ of
                   ...
                   | $\mathcal{C}(F(y),$fn $w$ => $e)$ $r$ => let  $z'$ = $\mathcal{X}(z, F(z), F(w))$
                                             $v$ = $\mathcal{N}($fn $w$ => $e)$($r$, $z'$)
                                             $v'$ = $\mathcal{X}(v, F(last(e)), F(x))$
                                          in $v'$
                                          end
                   ...
      where there is one branch for each fn $w$ => $e \in F(y)$ and $z'$, $v$, and $v'$ are fresh.
   d) $[\![C\ y]\!]_x$ = let  $y'$ = $\mathcal{X}(y, F(y), F(C))$
                      $r$ = $C\ y'$
                  in $r$
                  end
      where $y'$ and $r$ are fresh variables.
   e) $[\![$case $y$ of ... | $C\ z$ => $e$ | ...$]\!]_x$ =
      case $y$ of
         ...
         | $C\ z$ => let  $r$ = $[\![e]\!]$
                      $r'$ = $\mathcal{X}(r, F(last(e)), F(x))$
                  in $r'$
                  end
         ...
      where $r$, $r'$ are fresh variables.
   f) $[\![$( ... , $y$, ... )$]\!]_x$ = ( ... , $y$, ... )
   g) $[\![$#$i\ y]\!]_x$ = #$i\ y$
   h) $[\![$raise $y]\!]_x$ = raise $y$
   i) $[\![e_1$ handle $z$ => $e_2]\!]_x$ =  let $y_1$= $[\![e_1]\!]$
                                          $y_2$= $\mathcal{X}(y_1, F(last(e_1)), F(x))$
                                      in $y_2$ end
                                      handle $z$ => let $y_3$= $e_2$
                                                        $y_4$= $\mathcal{X}(y_2, F(last(e_2)), F(last(x)))$
                                                    in $y_4$ end
```

Fig. 4. Closure conversion of expressions.

4.1 Practical Issues

Although for a simple type system we must express coercions as a case expression with each arm simply changing the constructor (and the type) representing the closure, it is easy to pick an underlying representation for these datatypes so that no machine code actually has to be generated. In terms of the underlying memory objects, all coercions are the identity. If these datatypes are all

We define $\mathcal{X} : \mathit{Var} \times \mathit{AVal} \times \mathit{AVal} \to \mathit{Bind}$ by cases on abstract values. (Note, $\mathcal{X}(x, a, a')$ is only defined when $a \leq a'$.)

1. if $a = a'$ then $\mathcal{X}(x, a, a') = x$.
2. $\mathcal{X}(x, (\ldots,\ a_i,\ \ldots), (\ldots,\ a'_i,\ \ldots)) =$

```
let  ...
     $y_i$ = #$i$ $x$
     $y'_i$ = $\mathcal{X}(y_i, a_i, a'_i)$
     ...
     $z'$ = (..., $y'_i$, ...)
in $z'$
end
```

   where $z', \ldots, y_i, y'_i, \ldots$ are fresh variables.

3. $\mathcal{X}(x, L, L') =$

```
case $x$ of
   ...
 | $\mathcal{C}(L,$ fn $x$ => $e)$ $r$ => $\mathcal{C}(L',$ fn $x$ => $e)$ $r$
   ...
```

   where there is one branch for each `fn` x `=>` $e \in L$.

Fig. 5. The coercion function.

represented as a tag word (whose only function is to distinguish between the summands forming the datatype) followed by some fixed representation of the value being carried by that summand, then the only thing which might be changed by the coercion function is the tag word. It is thus easy to pick the tags so that they also don't change (for instance, use the address for the code of the procedure). However, we do not do this in MLton. As shown in Section 6, dynamic counts indicate coercions are so rare that their cost is unimportant. The advantage of allowing the coercions to change representations is that one can choose specialized representations for environment records.

The closure conversion algorithm is designed to be safe-for-space [1]. Note that each closure record is destructed at the beginning of each first order function. The alternative of replacing each reference to a closed-over variable with a selection from the closure record violates space safety because it keeps the entire record alive. Another possible violation is rule 2c, which can turn a tail-call into a non-tail-call by requiring a coercion after the call. However, since each such coercion corresponds to a step up the lattice of abstract values which is of finite height, the space usage of the program can only increase by a constant factor.

Finally, it is possible to share all of the dispatches generated for calls to a given set of λ-expressions. However, MLton does not do this, since it has not been necessary for performance.

5 Example

Consider the example in Figure 6.

```
let f = fn a => fn b => a
    g = fn c => fn d => d
    h = case ... of
             ...⇒ f
             ...⇒ g
    m = h 13
    _ = m 7
```

(a) Source program

$F(\mathtt{f}) = \{\mathtt{fn\ a}\}$
$F(\mathtt{g}) = \{\mathtt{fn\ c}\}$
$F(\mathtt{h}) = \{\mathtt{fn\ a}, \mathtt{fn\ c}\}$
$F(\mathtt{m}) = \{\mathtt{fn\ b}, \mathtt{fn\ d}\}$

(b) Flow

```
datatype t1 = C1 of unit  (* fn a *)
datatype t2 = C2 of int   (* fn b *)
datatype t3 = C3 of unit  (* fn c *)
datatype t4 = C4 of unit  (* fn d *)
datatype t5 = C5 of unit(* fn a ∈ F(h) *)
            | C6 of unit(* fn c ∈ F(h) *)
datatype t6 = C7 of int (* fn b ∈ F(m) *)
            | C8 of unit(* fn d ∈ F(m) *)
fun F (r, a) = C2 a     (* fn a *)
fun F' (r, b) =
  let a = #1 r in a end (* code: fn b *)
fun G (r, c) = C4 ()    (* code: fn c *)
fun G' (r, d) = d       (* code: fn d *)
val f = C1()
val g = C3()
val h = case ... of
            ...⇒ case f of C1 r ⇒ C5 r
            ...⇒ case g of C3 r ⇒ C6 r
val m = case h of
   C5 r ⇒(case F (r, 13) of C2 r ⇒ C7 r)
  | C6 r ⇒(case G (r, 13) of C4 r ⇒ C8 r)
val _ = case m of
       C7 r ⇒ F' (r, 7)
     | C8 r ⇒ G' (r, 7)
```

(c) Target program

Fig. 6. Example.

The source appears in part (a), the 0CFA flow is in part (b), and the result of closure conversion appears in part (c). We use `fn a` to represent the entire λ-expression beginning with `fn a`. Consider the translation of the last expression, the call to `m`. Since `m` may be bound to a procedure corresponding to `fn b` or `fn d`, the call must dispatch appropriately. For the expression which defines `h`, each branch of the `case`-expression must coerce a procedure corresponding to a known λ-expression to one which is associated with an element of {`fn a`, `fn c`}. In the expression defining `m`, both a dispatch and a coercion occur: first a dispatch based on the λ-expression which provides the code for the `h` is required. Then, each arm of this case expression must coerce the result (a function with known code) to one associated with either `fn b` or `fn d`.

6 Experiments

We have have implemented the algorithm as part of MLton, a whole-program compiler for Standard ML. MLton does not support separate compilation, and takes advantage of whole program information in order to perform many optimizations. Here, we give a brief overview of the relevant compiler passes and intermediate languages. First, MLton translates the input SML program into an explicitly-typed, polymorphic intermediate language (XML)[8]. XML does not have any module level constructs. All functor applications are performed at compile-time[6], and all uses of structures and signatures are eliminated by moving declarations to the top-level and appropriately renaming variables. Next, MLton translates the XML to SXML (a simply-typed language) by *monomorphisation*, eliminating all uses of polymorphism by duplicating each polymorphic expression for each monotype at which it is used. After monomorphisation, small higher-order functions are duplicated; a size metric is used to prevent excessive code growth. MLton then performs flow analysis as described in Section 3 on the resulting SXML, and closure converts procedures to FOL (a first-order simply-typed language) via the algorithm described in Section 4. After a series of optimizations (e.g., inlining, tuple flattening, redundant argument elimination, and loop invariant code motion), the FOL program is translated to a C program, which is then compiled by `gcc`. Like [22], a trampoline is used to satisfy tail-recursion. To reduce trampoline costs, multiple FOL procedures may reside in the same C procedure; a dispatch on C procedure entry jumps to the appropriate code [7].

To demonstrate the practicality of our approach, we have measured its impact on compile time and code size for benchmarks with sizes up to 75K lines. Among the benchmarks, `knuth-bendix`, `life`, `lexgen`, `mlyacc`, and `simple` are standard [1]; `ratio-regions` is integer intensive; `tensor` is floating-point intensive, and `count-graphs` is mostly symbolic[2]. `MLton` is the compiler itself, and `kit` is the ML-kit [25,24]. The benchmarks were executed on a 450 MHz Intel Xeon with 1 GB of memory.

In Table 1, we give the number of lines of SML for each benchmark, along with compile times both under SML/NJ (version 110.9.1)[3] and MLton. The number of lines does not include approximately 8000 lines of basis library code that MLton prefixes to each program. The compile time given for SML/NJ is the time to batch compile the entire program. In order to improve the performance of the code generated by SML/NJ, the entire program is wrapped in a `local` declaration whose body performs an `exportFn`. For MLton, we give the total compile time, the time taken by flow analysis and closure conversion, and the percentage of compile time spent by `gcc` to compile the C code.

The flow analysis times are shorter than previous work [2,10,4] would suggest, for several reasons. First, the sets of abstract values are implemented using hash

[2] `ratio-regions` was written by Jeff Siskind (`qobi@research.nj.nec.com`), `tensor` was written by Juan Jose Garcia Ripoll (`worm@arrakis.es`), and `count-graphs` was written by Henry Cejtin (`henry@clairv.com`).

[3] Except for the `kit` which is run under SML/NJ version 110.0.3 because 110.9.1 incorrectly rejects the `kit` as being ill-typed.

consing and the binary operations (in particular set union) are cached to avoid re-computation. Second, because of monomorphisation, running 0CFA on SXML is equivalent to the polyvariant analysis given in [12]. Thus, it is more precise than 0CFA performed directly on the (non-monomorphised) source alone, and hence fewer set operations are performed. Third, the analysis only tracks higher-order values. Finally, the analysis is less precise for datatypes than the usual *birthplace*[13] approach (see rules 4 and 5a in Figure 3). Also, unlike earlier attempts to demonstrate the feasibility of 0CFA [20] which were limited to small programs or intramodule analysis, our benchmarks confirm that flow analysis is practical for programs even in excess of 50K lines.

MLton compile-times are longer than SML/NJ. However, note that the ratio of MLton's to SML/NJ's compile-time does not increase as program size increases. We believe MLton's compile-time is in practice linear. In fact, gcc is a major component of MLton's compile-time, especially on large programs. We expect a native back-end to remove much of this time.

Table 2 gives various dynamic counts for these benchmarks to quantify the cost of closure conversion. To make the presentation tractable, the entries are in millions per second of the running time of the program. Nonzero entries less than .01 are written as ˜0. SXML Known and Unknown measure the number of known and unknown procedure calls identified in the SXML program using only syntactic heuristics [1]. FOL Known indicates the number of known procedure calls remaining in the FOL program after flow analysis and all optimizations on the FOL program have been performed. The difference between SXML and FOL Known is due to inlining and code simplificaton. Dispatch indicates the number of case expressions introduced in the FOL program to express procedure calls where the flow set is not a singleton. Thus, the difference between Dispatch and Unknown gives a rough measure of the effectiveness of flow analysis above syntactic analyses in identifying the procedures applied at call-sites. Finally,

Table 1. Program sizes (lines) and compile times (seconds).

	lines		MLton			
Program	SML	SML/NJ	Total	Flow	Convert	gcc%
count-graphs	204	1.2	4.02	.01	.25	38%
kit	73489	1375.75	2456.39	1.34	27.96	82%
knuth-bendix	606	2.7	6.55	.01	.32	47%
lexgen	1329	4.5	19.52	.03	.78	53%
life	161	.9	3.2	.01	.16	41%
MLton	47768	637.5	1672.0	1.94	33.84	81%
mlyacc	7297	30.1	144.86	.10	2.34	38%
ratio-regions	627	2.2	6.22	.01	.35	34%
simple	935	4.7	34.11	.04	.87	54%
tensor	2120	9.7	10.12	.03	.32	30%
tsp	495	.8	3.56	.01	.22	30%

Table 2. Dynamic counts (millions/second).

	SXML		FOL		
Program	Known	Unknown	Known	Dispatch	Coerce
`count-graphs`	60.2	˜0	1.0	0	0
`kit`	13.1	.11	5.8	.02	˜0
`knuth-bendix`	28.8	˜0	11.3	˜0	0
`lexgen`	63.4	2.68	15.4	˜0	0
`life`	28.4	0	22.3	0	0
`MLton`	14.5	.48	5.2	.34	.01
`mlyacc`	37.5	.03	10.6	˜0	0
`ratio-regions`	119.4	0	14.3	0	0
`simple`	34.2	.26	6.2	.26	0
`tensor`	140.6	˜0	7.6	˜0	0
`tsp`	34.5	˜0	3.4	˜0	0

Coerce indicates the number of coercions performed on closure tags to ensure that the closure's type adheres to the appropriate flow set.

For most benchmarks, monomorphisation, and aggressive syntactic inlining make most calls known. However, for several of the benchmarks, there still remain a significant number of unknown calls. Flow analysis uniformly helps in reducing this number. Indeed, the number of dispatches caused by imprecision in the analysis is always less than 5% of the number of calls executed. Notice also that the number of coercions performed is zero for the majority of the benchmarks; this means imprecision in the flow analysis rarely results in unwanted merging of closures with different representations.

Table 3 gives runtime results for both SML/NJ and MLton. Of course, because the two systems have completely different compilation strategies, optimizers, backends, and runtime systems, these numbers do not isolate the performance of our closure conversion algorithm. However, they certainly demonstrate its feasibility.

Table 3. Runtimes (in seconds) and ratio of SML/NJ to MLton.

Program	SML/NJ (sec)	MLton (sec)	NJ/MLton
`count-graphs`	28.8	11.9	2.40
`kit`	27.5	30.9	.89
`knuth-bendix`	44.1	15.2	2.90
`lexgen`	52.7	31.8	1.66
`life`	51.5	54.2	.95
`MLton`	198.7	101.3	1.96
`mlyacc`	43.4	20.6	2.11
`ratio-regions`	122.5	18.9	6.48
`simple`	25.3	18.4	1.38
`tensor`	154.4	19.8	7.78
`tsp`	191.7	25.4	7.54

7 Related Work and Conclusions

Closure conversion algorithms for untyped target languages have been explored in detail [1,21]. Algorithms that use a typed target language, however, must solve the problem created when procedures of the same type differ in the number and types of their free variables. Since closure conversion exposes the types of these variables through an explicit environment record, procedures having the same source-level type may compile to closures of different types. Minamide et al. [16] address this problem by defining a new type system for the target language that uses an existential type to hide the environment component of a closure record in the closure's type, exposing the environment only at calls. Unfortunately, the target language is more complex than the simply-typed λ-calculus and makes it difficult to express control-flow information. For example, the type system prevents expressing optimizations that impose specialized calling conventions for different closures applied at a given call-site.

An alternative to Minamide et al.'s solution was proposed by Bell et al. [3]. Their approach has the benefit of using a simply-typed target language, but does not express control-flow information in the target program. Inspired by a technique first described by Reynolds [19], they suggest representing closures as members of a datatype, with one datatype for each different arrow type in the source program. Tolmach and Oliva [26] extend Bell et al. by using a weak monovariant flow analysis based on type inference [11]. They refine the closure datatypes so that there is one datatype for each equivalence class of procedures as determined by unification. Although their approach does express flow analysis in a simply-typed target language, it is restricted to flow analyses based on unification. We differ from these approaches by using datatype coercions to produce a simply-typed target program and in our use of 0CFA.

Dimock et al. [5] describe a flow-directed representation analysis that can be used to drive closure conversion optimizations. Flow information is encoded in the type system through the use of intersection and union types. Like our work, their system supports multiple closure representations in a strongly-typed context. However, they support only a limited number of representation choices, and rely critically on a more complex type system to express these choices. Our work also uses flow information to make closure representation decisions, but does so within a simply-typed λ calculus.

Palsberg and O'Keefe[18] define a type system that accepts the same set of programs as 0CFA viewed as safety analysis. Their type system is based on simple types, recursive types, and subtyping. Although they do not discuss closure conversion, our coercions correspond closely to their use of subtyping. By inserting coercions, we remove the need for subtyping in the target language, and can use a simpler language based on simple types, sum types, and recursive types.

Our work is also related to other compiler efforts based on typed intermediate representations [23,14]. Besides helping to verify the implementation of compiler optimizations by detecting transformations that violate type safety, typed intermediate languages expose representations (through types) useful for code generation. For example, datatypes in the target language describe environment representations as determined by flow analysis on the source language. Types therefore provide a useful bridge to communicate information across different compiler passes.

The results of our flow-directed closure conversion translation in MLton demonstrate the following:

1. First-order simply-typed intermediate languages are an effective tool for compilation of languages like ML.
2. The coercions and dispatches introduced by flow-directed closure conversion have negligible runtime cost.
3. Contrary to folklore, OCFA can be implemented to have negligible compile-time cost, even for large programs.

References

1. Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.
2. J. Michael Ashley. A practical and flexible flow analysis for higher-order languages. In *ACM Symposium on Principles of Programming Languages*, pages 195–207, January 1996.
3. Jeffrey M. Bell, Françoise Bellegarde, and James Hook. Type-driven defunctionalization. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 25–37, Amsterdam, The Netherlands, 9–11 June 1997.
4. Greg Defouw, David Grove, and Craig Chambers. Fast interprocedural class analysis. In *ACM Symposium on Principles of Programming Languages*, pages 222–236, January 1998.
5. Allyn Dimock, Robert Muller, Franklyn Turback, and J.B. Wells. Strongly-typed flow-directed representation transformations. In *International Conference on Functional Programming*, June 1997.
6. Matrin Elsman. Static interpretation of modules. In *International Conference on Functional Programming*, September 1999.
7. Marc Feeley, James Miller, Guillermo Rozas, and Jason Wilson. Compiling higher-order languages into fully tail-recursive portable c. Technical Report Technical Report 1078, Department of Computer Science, University of Montreal, 1997.
8. Robert Harper and John C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252, April 1993.
9. Nevin Heintze. Set-based analysis of ML programs. In *ACM Conference on LISP and Functional Programming*, pages 306–317, 1994.
10. Nevin Heintze and David A. McAllester. Linear-time subtransitive control flow analysis. In *Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation (PLDI)*, pages 261–272, Las Vegas, Nevada, 15–18 June 1997. *SIGPLAN Notices* 32(5), May 1997.
11. Fritz Henglein. Simple closure analysis. Technical Report D-193, Department of Computer Science, University of Copenhagen, March 1992.
12. Suresh Jagannathan, Stephen T. Weeks, and Andrew K. Wright. Type-directed flow analysis for typed intermediate languages. In *International Static Analysis Symposium*, September 1997.

13. Neil D. Jones and Stephen S. Muchnick. *Flow Analysis and Optimization of LISP-like Structures*, chapter 4, pages 102–131. Prentice–Hall, 1981.
14. Simon L. Peyton Jones, John Launchbury, Mark Shields, and Andrew Tolmach. Bridging the gulf: a common intermediate language for ML and Haskell. In *ACM Symposium on Principles of Programming Languages*, pages 49–51, January 1998.
15. Robin Milner, Mads Tofte, Robert Harper, and David B. Macqueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.
16. Yasuhiko Minamide, J. Gregory Morrisett, and Robert Harper. Typed closure conversion. In *ACM Symposium on Principles of Programming Languages*, pages 271–283, St. Petersburg, FL, January 1996.
17. Jens Palsberg. Closure analysis in constraint form. *ACM Transactions on Programming Languages and Systems*, 17(1):47–62, January 1995.
18. Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. *ACM Transactions on Programming Languages and Systems*, 17(4):576–599, July 1995. Preliminary version in Principles of Programming Languages, pages 367–378, January 1995.
19. John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM Annual Conference*, pages 717–740, 1972.
20. Manuel Serrano and Pierre Weis. $1 + 1 = 1$: an optimizing Caml compiler. In *Workshop on ML and its applications*, Orlando, Florida, June 1994. ACM SIGPLAN. Also appears as INRIA RR-2301.
21. Zhong Shao and Andrew W. Appel. Space-efficient closure representations. In *ACM Conference on LISP and Functional Programming*, pages 150–161, Orlando, FL, Jun 1994.
22. David Tarditi, Anurag Acharya, and Peter Lee. No assembly required: Compiling Standard ML to C. *ACM Letters on Programming Languages and Systems*, June 1992. Appears as CMU-CS-90-187.
23. David Tarditi, J. Gregory Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *ACM Conference on Programming Language Design and Implementation*, May 1996.
24. Mads Tofte and Lars Birkedal. A region inference algorithm. *Transactions on Programming Languages and Systems*, 20(4):724–767, 1998.
25. Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Olesen, Peter Sestoft, and Peter Bertelsen. Programming with regions in the ML Kit. Technical Report Technical Report DIKU-TR-97/12, Department of Computer Science, University of Copenhagen, 1997.
26. Andrew Tolmach and Dino Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, July 1998.

Directional Type Checking for Logic Programs: Beyond Discriminative Types

Witold Charatonik

Max-Planck-Institut für Informatik
Im Stadtwald, 66123 Saarbrücken, Germany
www.mpi-sb.mpg.de/~witold
and
University of Wrocław, Poland

Abstract. Directional types form a type system for logic programs which is based on the view of a predicate as a *directional procedure* which, when applied to a tuple of input terms, generates a tuple of output terms. It is known that directional-type checking wrt. arbitrary types is undecidable; several authors proved decidability of the problem wrt. discriminative regular types. In this paper, using techniques based on tree automata, we show that directional-type checking for logic programs wrt. general regular types is DEXPTIME-complete and fixed-parameter linear. The latter result shows that despite the exponential lower bound, the type system might be usable in practice.

Keywords: types in logic programming, directional types, regular types, tree automata.

1 Introduction

It is commonly agreed that types are useful in programming languages. They help understanding programs, detecting errors or automatically performing various optimizations. Although most logic programming systems are untyped, a lot of research on types in logic programming has been done [33].

Regular types. Probably the most popular approach to types in logic programming uses *regular* types, which are sets of ground terms recognized by finite tree automata (in several papers, including this one, this notion is extended to non-ground terms). Intuitively, regular sets are finitely representable sets of terms, just as in case of regular sets of words, which are finitely representable by finite word automata.

Actually, almost all type systems occurring in the literature are based on some kinds of regular grammars which give a very natural (if not the only) way to effectively represent interesting infinite collections of terms that denote e.g. lists or other recursive data structures. Of course some of them use extensions of regular sets with non-regular domains like numbers (see the discussion in Section 3.3), nonground terms (where all types restricted to ground terms are

G. Smolka (Ed.): ESOP/ETAPS 2000, LNCS 1782, pp. 72–87, 2000.

regular), polymorphism (where all monotypes, that is ground instances of polymorphic types, are regular). Very few systems go further beyond regular sets (and usually say nothing about computability issues for combining types).

Prescriptive and descriptive approaches. There are two main streams in the research on types in logic programming. In the prescriptive stream the user has to provide type declarations for predicates; these declarations form an integral part of the program. The system then checks if the program is well-typed, that is, if the type declarations are consistent. The present paper falls into the prescriptive stream.

In the descriptive stream the types are inferred by the system and used to describe semantic properties of untyped programs. The basic idea here is to over-approximate the least model of a given program by a regular set. This approach can be found in particular in [30,39,25,19,16,26,21,17], or, using a type-graph representation of regular sets (a type graph may be seen as a deterministic top-down tree automaton), in [29,28]. An advantage of the descriptive approach is that there is no need for type declarations; a disadvantage is that the inferred types may not correspond to the intent of the programmer.

The approximation of the least model of the program by a regular set is often not as precise as one would expect. A typical example here is a clause $append([], L, L)$. Most of the systems approximate the success set of this clause by the set of triples $\langle [], x, y \rangle$ where x and y are any terms and thus loose the information that a second and third argument are of the same type. To overcome this problem, [24,20] introduced approximations based on magic-set transformation of the input program. It was observed in [12] that types of the magic-set transformation of a program coincide with directional types of the initial program as they appear in [35,8,4,2,1,3,6,5,7].

Directional types. Directional types form a type system for logic programs which is based on the view of a predicate as a *directional procedure* which, when applied to a tuple of input terms, generates a tuple of output terms. They first occurred in [35] as predicate profiles and in [8] as mode dependencies. Our use of the terminology “directional type” stems from [1].

Discriminative types. In most type systems for logic programs that are based on regular types, the types are restricted to be *discriminative* (equivalently, path-closed or tuple-distributive or recognizable by deterministic top-down tree automata). The reason for that is probably a hope for better efficiency or conceptual simplicity of such approach. Unfortunately, discriminative sets are closed under neither union nor complementation. A union of two discriminative sets is then approximated by a least discriminative set that contains both of them, but then the distributivity laws for union and intersection do not hold anymore. This is very unintuitive and has lead already to several wrong results. One of the results of this paper is that in the context of directional types the restriction to discriminative types, at least theoretically, does not pay: the exponential lower bound for the discriminative case is matched by the exponential upper bound for the general case. In fact, as shown in [14], even stronger restriction to unary types (where all paths in a tree are disconnected from each other, see e.g. [39])

is not worth it, since type checking problem (even for non-directional types) remains hard for DEXPTIME.

Complexity of type checking. The exponential lower bound of the type-checking problem looks quite discouraging at the first view. A closer look into the proof of the lower bound shows that the program used there was very short while the types were quite large (the encoding of the Turing-machine computation was done in the types, not in the program). The situation in practice looks usually quite opposite: the type-checking problems are usually applied to large programs and rather small types. This lead us to study the parameterized complexity of the problem, where the parameter is given by the length of the type to be checked. We obtained here quite a nice result: the problem is fixed-parameter linear, which means that for a fixed family of types the type-checking problem can be decided in time linear in the size of the program. This shows that there is a good potential for the type system to be practical. A similar phenomenon is known already for the functional language ML where bad theoretical lower bounds do not match good practical behaviour of the time system. The explanation was given by Henglein [27] who showed that typability by types of size bounded by constant is polynomial time decidable.

Related work. It is pointed out in [1] that the type checking problem is undecidable for arbitrary directional types. Therefore Aiken and Lakshman restrict themselves to regular directional types. Although their algorithm for automatic type checking is sound for general regular types, it is sound and complete only for discriminative ones. It is based on solving negative set constraints and thus runs in nondeterministic exponential time. Another algorithm (without complexity analysis) for type-checking for discriminative directional types is given in [5]. In [12] it is proved that directional type checking wrt. discriminative types is DEXPTIME-complete and an algorithm for *inferring* (regular, not necessarily discriminative) directional types is given.

Rychlikowski and Truderung [36] proposed recently a system of polymorphic directional types. The types there are incomparable with ours: on one hand they are more general because of the use of the polymorphism; on the other hand they are even more restricted than regular discriminative types (e.g. they are not able to express lists of an even length). The authors presented a type-checking algorithm working in DEXPTIME, but probably the most interesting feature of this system is the inference of so-called main type of a predicate — the type that provides a compact representation of all types of the predicate.

Our results. The methods used in the mentioned papers are not strong enough to prove the decidability of directional type checking wrt. general regular types. In this paper, using tree-automata techniques, we prove that this problem is decidable in DEXPTIME, which, together with the result from [12] stating DEXPTIME-hardness, establishes DEXPTIME-completeness of the problem. Moreover, we show that the problem is fixed-parameter linear – our procedure is exponential in the size of the input types, but linear in the size of the program. This improves the results by Aiken and Lakshman [1], Boye [5], and Charatonik and Podelski [12], where decidability is restricted to discriminative types.

Decidability of directional type checking wrt. general regular types has already a quite long history. It was first proved [11] by a reduction to the encompassment theory [9]. The result was not satisfactory because of the complexity of the obtained procedure: several (around five) times exponential. In [10] we found another solution based on a different kind of automata and reduced the complexity to NEXPTIME. The proof presented here is a refinement of the argument from [10].

2 Preliminaries

If Σ is a signature (that is, set of function symbols) and Var is a set of variables then T_Σ is the set of ground terms and $T_{\Sigma(\mathsf{Var})}$ is the set of non-ground terms over Σ and Var. We write $\mathsf{Var}(t)$ for the set of variables occurring in the term t. The notation $|S|$ is used, depending on the context, for the cardinality of the set S or for the size of the object S (that is, the length of the word encoding S).

2.1 Tree Automata

The methods we use are based on tree-automata techniques. Standard techniques as well as all well-known results that we mention here can be found e.g. in [22,15,38]. Below we recall basic notions in this area.

Definition 1 (Tree automaton). *A tree automaton is a tuple $\mathcal{A} = \langle \Sigma, Q, \Delta, F \rangle$ where Σ, Q, Δ, F are finite sets such that*

- *Σ is a signature,*
- *Q is a finite set of states,*
- *Δ is set of transitions of the form $f(q_1, \ldots, q_n) \to q$ where $f \in \Sigma$, $q, q_1, \ldots, q_n \in Q$ and n is the arity of f,*
- *$F \subseteq Q$ is a set of final states.*

The automaton $\mathcal{A}$ is called

- *bottom-up deterministic, if for all $f \in \Sigma$ and all sequences $q_1, \ldots, q_n \in Q$ there exists at most one $q \in Q$ such that $f(q_1, \ldots, q_n) \to q \in \Delta$,*
- *top-down[1] deterministic if $|F| = 1$ and for all $f \in \Sigma$ and all $q \in Q$ there exists at most one sequence $q_1, \ldots, q_n \in Q$ such that $f(q_1, \ldots, q_n) \to q \in \Delta$,*
- *complete, if for all $f \in \Sigma$ and all sequences $q_1, \ldots, q_n \in Q$ there exists at least one $q \in Q$ such that $f(q_1, \ldots, q_n) \to q \in \Delta$.*

A tree automaton $\mathcal{A} = \langle \Sigma, Q, \Delta, F \rangle$ translates to a logic program containing a clause $q(f(x_1, \ldots, x_n)) \leftarrow q_1(x_1), \ldots, q_n(x_n)$ for each transition $f(q_1, \ldots, q_n) \to q \in \Delta$, where one is interested only in queries about the predicates in F.

[1] Intuitively, a top-down automaton reads trees top-down, and thus F is here the set of initial (not final) states.

Definition 2 (Run). *A run of a tree automaton $\mathcal{A} = \langle \Sigma, Q, \Delta, F\rangle$ on a tree $t \in T_\Sigma$ is a mapping ρ assigning a state to each occurrence of a subterm $f(t_1, \ldots, t_n)$ of t such that*

$$f(\rho(t_1), \ldots, \rho(t_n)) \to \rho(f(t_1, \ldots, t_n)) \in \Delta.$$

A run ρ on t is successful *if $\rho(t) \in F$.*

Sometimes we will refer to runs over terms in $T_{\Sigma \cup Q}$. We then extend the definition above by putting $\rho(q) = q$ for all states in Q.

If there exists a successful run on a tree t then we say that the automaton accepts, or recognizes, t. The set of all trees accepted by an automaton $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is called the *language* of the automaton $\mathcal{A}$, or the set *recognized* by this automaton. A set of trees is called *regular* if it is recognized by some tree automaton.

A state q of the automaton $\mathcal{A}$ is called [bottom-up] *reachable* if there exists a tree $t \in T_\Sigma$ and a run ρ of $\mathcal{A}$ on t such that $\rho(t) = q$.

It is well-known (cf. [22,15,38]) that regular languages are closed under Boolean operations: one can effectively construct in polynomial time an automaton that recognizes union or intersection of given two regular languages, and in exponential time an automaton that recognizes complement. In polynomial time one can compute the set of reachable states and thus test emptiness of the language recognized by a given automaton. In exponential time one can determinize an automaton, that is, construct a bottom-up deterministic automaton that recognizes the same set. Tree automata are not top-down determinizable.

Example 1. Consider the automaton $\mathcal{A} = \langle \{a, f\}, \{q_0, q_1, q\}, \{a \to q_0, a \to q_1, f(q_0, q_1) \to q\}, \{q\}\rangle$. The run that assigns q_0 to the first occurrence of a, q_1 to the second occurrence of a and q to $f(a, a)$ is a successful run of $\mathcal{A}$ on $f(a, a)$. The automaton $\mathcal{A}$ is top-down deterministic, is not bottom-up deterministic, and is not complete.

2.2 Directional Types

By logic programs we mean definite horn-clause programs (pure Prolog programs). For the sake of simplicity we assume that all predicate symbols occurring in this paper are unary (there is no loss of generality since function symbols may be used to form tuples). The set of predicate symbols occurring in a program $\mathcal{P}$ is denoted $\mathsf{Pred}(\mathcal{P})$ or simply Pred if $\mathcal{P}$ is clear from the context. For a program $\mathcal{P}$, $lm(\mathcal{P})$ denotes its least model. For $p \in \mathsf{Pred}(\mathcal{P})$ we define $[\![p]\!]_\mathcal{P} = \{t \mid p(t) \in lm(\mathcal{P})\}$.

A *type* is a set of terms closed under substitution [2]. A *ground type* is a set of ground terms (i.e., trees), and thus a special case of a type. A term t has type T, in symbols $t{:}T$, if $t \in T$. A *type judgment* is an implication $t_1{:}T_1 \wedge \ldots \wedge t_n{:}T_n \to t_0{:}T_0$. We say that such a judgment *holds* if the implication $t_1\theta \in T_1 \wedge \ldots \wedge t_n\theta \in T_n \to t_0\theta \in T_0$ is true for all term substitutions $\theta : \mathsf{Var} \to T_{\Sigma(\mathsf{Var})}$.

We recall that a set of ground terms is regular if it can be defined by a finite tree automaton (or, equivalently, by a ground set expression as in [1] or a regular grammar as in [16]). The definition below coincides with the types used in [1], it extends the definition from [16] by allowing non-ground types, and is equivalent to the definition from [5].

Definition 3 (Regular type). *A type is* regular *if it is of the form $Sat(T)$ for a regular set T of ground terms, where the set $Sat(T)$ of terms satisfying T is the type*

$$Sat(T) = \{t \in T_{\Sigma(\mathsf{Var})} \mid \theta(t) \in T \textit{ for all ground substitutions } \theta : \mathsf{Var} \to T_{\Sigma}\}.$$

Definition 4 (Directional type of a program [8,1]). *A* directional type *of a program $\mathcal{P}$ is a family*

$$\mathcal{T} = (I_p \to O_p)_{p \in \mathsf{Pred}}$$

assigning to each predicate p of $\mathcal{P}$ an input type I_p and an output type O_p such that, for each clause $p_0(t_0) \leftarrow p_1(t_1), \ldots, p_n(t_n)$ of $\mathcal{P}$, the following type judgments hold.

$$\begin{aligned} t_0 : I_{p_0} &\to t_1 : I_{p_1} \\ t_0 : I_{p_0} \wedge t_1 : O_{p_1} &\to t_2 : I_{p_2} \\ &\vdots \\ t_0 : I_{p_0} \wedge t_1 : O_{p_1} \wedge \ldots \wedge t_{n-1} : O_{p_{n-1}} &\to t_n : I_{p_n} \\ t_0 : I_{p_0} \wedge t_1 : O_{p_1} \wedge \ldots \wedge t_n : O_{p_n} &\to t_0 : O_{p_0} \end{aligned}$$

We then also say that $\mathcal{P}$ is well-typed wrt. $\mathcal{T}$.

Following [1] we define that a query $q_1(t_1), \ldots, q_n(t_n)$ is well-typed if for all $1 \le j \le n$ the judgment $\bigwedge_{1 \le k < j} t_i : O_{q_i} \to t_j : I_{q_j}$ holds. It is then easy to see that "well-typed programs do not go wrong" as defined in [31]. Namely, an application of one step of SLD-resolution to a well-typed query results always in a new well-typed query. This does not say, however, anything about whether the query succeeds, fails or loops.

The definition above refers to the operational semantics of logic programs based on left to right execution. There is also a more declarative (cf. [32], see also Theorem 1) intuition behind it: Intuitively, the judgments say that if a query has the correct input type and its call terminates successfully, then the computed answer has the correct output type.

Definition 5 (Type checking). *The type-checking problem is to decide for a given program $\mathcal{P}$ and directional type $\mathcal{T}$, whether $\mathcal{P}$ is well-typed wrt. $\mathcal{T}$.*

A program can have many directional types. For example, consider the predicate *append* defined by

$$\begin{aligned} &append([], L, L). \\ &append([X|Xs], Y, [X|Z]) \leftarrow append(Xs, Y, Z). \end{aligned}$$

We can give this predicate the directional type $(list, list, \top) \to (list, list, list)$, where *list* denotes the set of all lists and $\top$ is the set of all terms, but also

$(\top, \top, \mathit{list}) \rightarrow (\mathit{list}, \mathit{list}, \mathit{list})$, as well as $(\top, \top, \top) \rightarrow (\top, \top, \top)$. (Recall that *append* is seen as a unary predicate here, and $(\mathit{list}, \mathit{list}, \mathit{list})$ is a set of terms which are triples of lists.) A predicate defined by a single fact $p(X)$ has a directional type $\tau \rightarrow \tau$ for all types τ.

Example 2. We show that $(\mathit{list}, \mathit{list}, \top) \rightarrow (\mathit{list}, \mathit{list}, \mathit{list})$ well-types the predicate *append* defined above. For the first clause, $\mathit{append}([], L, L)$, we have to show only one judgment, namely

$$\mathit{append}([], L, L) : (\mathit{list}, \mathit{list}, \top) \rightarrow \mathit{append}([], L, L) : (\mathit{list}, \mathit{list}, \mathit{list}).$$

The condition we have to check here is a tautology: the assumption that $[]$ is a member of *list* implies that $[]$ is a member of *list*, and the assumption that L is a member of both *list* and $\top$ implies that L is a member of *list*. For the second clause, $\mathit{append}([X|Xs], Y, [X|Z]) \leftarrow \mathit{append}(Xs, Y, Z)$ we have two judgments:

$$\mathit{append}([X|Xs], Y, [X|Z]) : (\mathit{list}, \mathit{list}, \top) \rightarrow \mathit{append}(Xs, Y, Z) : (\mathit{list}, \mathit{list}, \top),$$

and

$$\begin{aligned}\mathit{append}([X|Xs], Y, [X|Z]) : (\mathit{list}, \mathit{list}, \top), \mathit{append}(Xs, Y, Z) : (\mathit{list}, \mathit{list}, \mathit{list})\\ \rightarrow \mathit{append}([X|Xs], Y, [X|Z]) : (\mathit{list}, \mathit{list}, \mathit{list}).\end{aligned}$$

The first one follows from the observation that if $[X|Xs]$ is a list then Xs is a list. The second, from the observation that if Z is a list then $[X|Z]$ is a list.

A similar reasoning can be used to show that $((\mathit{list}, \mathit{list}, \top) \cup (\top, \top, \mathit{list})) \rightarrow (\mathit{list}, \mathit{list}, \mathit{list})$ well-types *append*. Then both $\mathit{append}([a, b, X], [c], L)$ and $\mathit{append}(X, Y, [a, b, c])$ are well-typed queries while $\mathit{append}([a], X, Y)$ is not.

We do not use discriminative types in this paper. We include the definition below to show what the contribution of the paper is. The notion of a path-closed set below originates from [22]. It is equivalent to other notions occurring in the literature: tuple-distributive [30,35], discriminative [1], or deterministic.

Definition 6 (Discriminative type). *A regular set of ground terms is called* path-closed *if it can be defined by a* deterministic *top-down tree automaton. A directional type is called* discriminative *if it is of the form*

$$(\mathit{Sat}(I_p) \rightarrow \mathit{Sat}(O_p))_{p \in \mathsf{Pred}},$$

where the sets I_p, O_p are path-closed.

A deterministic top-down tree automaton translates to a logic program which does not contain two different clauses with the same head (modulo variable renaming), e.g., $p(f(x_1, \ldots, x_n)) \leftarrow p_1(x_1), \ldots, p_n(x_n)$ and $p(f(x_1, \ldots, x_n)) \leftarrow p'_1(x_1), \ldots, p'_n(x_n)$. A discriminative set expression as defined in [1] translates to a deterministic finite tree automaton, and vice versa. That is, discriminative set expressions denote exactly path-closed regular sets. It is argued in [1] that discriminative set expressions are quite expressive and are used to express commonly used data structures. Note that lists, for example, can be defined by the program with the two clauses $list(cons(x, y)) \leftarrow list(y)$ and $list(nil)$.

There are, however, many regular types which are not discriminative. The simplest is the set $\{f(a, a), f(b, b)\}$. Another simple example of a regular but not path-closed set is given in Example 2: it is the set consisting of triples $\langle x, y, z\rangle$ where either x and y are lists and z is any term or x and y are any terms and z is a list (which is useful for typing of the predicate *append* used either for concatenating of the lists x and y or for splitting the list z).

The use of general regular types has also other advantages: it gives us overloading for free. For example, if an operator like + is used in addition of both integers and reals, the corresponding automaton may have simply both transitions $+(\mathsf{int}, \mathsf{int}) \rightarrow \mathsf{expr}$ and $+(\mathsf{real}, \mathsf{real}) \rightarrow \mathsf{expr}$.

Further motivation for studying regular but not discriminative types comes from program verification. Several papers, including [13,23,34] modeled transition systems as logic programs. In many cases safety properties can be tested by type checking: it is enough to prove that some predicates have types of the form $Goodstates \rightarrow Goodstates$ where *Goodstates* is a set which does not contain unsafe states. For example, if we reason about mutual exclusion of two concurrent processes, the set *Goodstates* could contain three terms: $state(noncritical, noncritical)$, $state(noncritical, critical)$ and $state(critical, noncritical)$. However, any discriminative set containing both terms $state(noncritical, critical)$ and $state(critical, noncritical)$ must also contain the term $state(critical, critical)$ and thus we cannot verify mutual exclusion within such a type system. It is known (cf. [13]) that regular (not limited to discriminative) sets can capture all temporal properties expressible in the logic CTL for all finite systems as well as for some infinite ones, like pushdown or some parameterized systems. Since most model-checkers are limited to finite-state systems, there is a good potential for applications of the logic-programming approach to the infinite case. But to apply a type system for verification we need the full power of regular sets.

3 Directional Type Checking

In this section we prove that the directional type checking for logic programs wrt. general regular types is DEXPTIME-complete and fixed-parameter linear.

We start with recalling a technique used in [12]. We transform the well-typedness condition in Definition 4 into a logic program $\mathcal{P}_{InOut}$ by replacing $t : I_p$ with the atom $p^{In}(t)$ and $t : O_p$ with $p^{Out}(t)$.

Definition 7 ($\mathcal{P}_{InOut}$, the type program for $\mathcal{P}$). *Given a program $\mathcal{P}$, the corresponding type program $\mathcal{P}_{InOut}$ defines an in-predicate p^{In} and an out-predicate p^{Out} for each predicate p of $\mathcal{P}$. Namely, for every clause $p_0(t_0) \leftarrow p_1(t_1), \ldots, p_n(t_n)$ in $\mathcal{P}$, $\mathcal{P}_{InOut}$ contains the n clauses defining in-predicates corresponding to each atom in the body of the clause,*

$$\begin{aligned}
&p_1^{In}(t_1) \leftarrow p_0^{In}(t_0)\\
&p_2^{In}(t_2) \leftarrow p_0^{In}(t_0),\, p_1^{Out}(t_1)\\
&\qquad\qquad\vdots\\
&p_n^{In}(t_n) \leftarrow p_0^{In}(t_0),\, p_1^{Out}(t_1), \ldots,\, p_{n-1}^{Out}(t_{n-1})
\end{aligned}$$

and the clause defining the out-predicate corresponding to the head of the clause,

$$p_0^{Out}(t_0) \leftarrow p_0^{In}(t_0),\, p_1^{Out}(t_1), \ldots,\, p_n^{Out}(t_n).$$

The program above is known in the literature as a magic-set transformation of the initial program $\mathcal{P}$. It was used (among other things) to obtain more precise information about answers computed by the program if the queries are restricted to some specific form. If we denote by $\mathcal{P}_{In}$ a program that defines some p^{In} predicates (intuitively, the queries to the program $\mathcal{P}$ are then restricted to those defined in the program $\mathcal{P}_{In}$) then it is easy to observe that

$$[\![p^{Out}]\!]_{\mathcal{P}_{In} \cup \mathcal{P}_{InOut}} = [\![p]\!]_{\mathcal{P}} \cap [\![p^{In}]\!]_{\mathcal{P}_{In} \cup \mathcal{P}_{InOut}},$$

which intuitively means that an atom $p^{Out}(t)$ is in the least model of the transformed program if and only if $p(t)$ is in the least model of the initial program and $p^{In}(t)$ is allowed as a query.

The following theorem is proved in [12]. Essentially, it says that a directional type of the form $\mathcal{T} = (Sat(I_p) \to Sat(O_p))_{p \in \mathsf{Pred}}$, for ground types $I_p, O_p \subseteq T_\Sigma$, satisfies required type judgments if and only if the corresponding directional *ground* type $\mathcal{T}_g = (I_p \to O_p)_{p \in \mathsf{Pred}}$ does.

Theorem 1 (Types and models of type programs). *A program $\mathcal{P}$ is well-typed wrt. the directional type*

$$\mathcal{T} = (Sat(I_p) \to Sat(O_p))_{p \in \mathsf{Pred}}$$

(with ground types I_p, O_p) if and only if the subset of the Herbrand base corresponding to $\mathcal{T}$,

$$\mathcal{M}_{\mathcal{T}} = \{p^{In}(t) \mid t \in I_p\} \cup \{p^{Out}(t) \mid t \in O_p\},$$

is a model of the type program $\mathcal{P}_{InOut}$.

Note that the theorem above connects directional types with arbitrary models of the type program, not only with the least model. Since every clause in this program contains occurrences of predicates p^{In} and there are no facts defining these predicates, the least model is empty, which corresponds to the trivial directional type $\emptyset \to \emptyset$ (and expresses that a program without input does not produce output). On the other extremity we have the whole Herbrand base, which is also a model of the type program and corresponds to the trivial type $\top \to \top$.

3.1 Exponential Upper Bound

Note that a subset of the Herbrand base is a model of a logic program if and only if it is a model of each clause of the program. Thus, as an immediate consequence of Theorem 1 above we obtain that the type-checking problem for directional types reduces to the following model-checking problem.

Problem 1. Given a clause $p_0(t_0) \leftarrow p_1(t_1), \ldots, p_n(t_n)$ and a family of regular sets $T_{p_0}, T_{p_1}, \ldots, T_{p_n}$, decide whether the set $\bigcup_{i=0}^{n} \{p_i(t) \mid t \in T_{p_i}\}$ is a model of the clause.

The problem above is closely related to another problem known in the theory of tree automata (in particular, for reasoning about ground reducibility, see [15]), namely, if for a given term t and regular set T there exists a ground instance of t in T. We use similar techniques to prove its decidability. However, since we want to carefully analyze its complexity, we find it easier to present a direct proof rather than to find a suitable reduction. For the decidability proof we need the following lemma.

Lemma 1. *Let $\mathcal{A}_i = \langle \Sigma, Q_i, \Delta_i, F_i \rangle$ for $i = 0, \ldots, n$ be tree automata with disjoint sets of states, and let $\# \notin \Sigma$ be a fresh function symbol of arity $n+1$. There exists a tree automaton $\mathcal{A} = \langle \Sigma \cup \{\#\}, Q, \Delta, F \rangle$ such that*

- *$\mathcal{A}$ is bottom-up deterministic, and*
- *all states of $\mathcal{A}$ are reachable, and*
- *$\mathcal{A}$ recognizes the set $\#(T_\Sigma - \mathcal{L}(\mathcal{A}_0), \mathcal{L}(\mathcal{A}_1), \ldots, \mathcal{L}(\mathcal{A}_n))$, and*
- *$\mathcal{A}$ can be effectively constructed from $\mathcal{A}_0, \ldots, \mathcal{A}_n$ in single exponential time.*

Proof. The idea of the proof below is to use standard complementation and determinisation methods to construct an automaton $\mathcal{A}' = \langle \Sigma \cup \{\#\}, Q', \Delta', F' \rangle$ that satisfies all conditions except reachability of states. The only problem here is that we have to complement and determinize at the same time to avoid a doubly-exponential blowup. Then we obtain $\mathcal{A}$ by removing non-reachable states from $\mathcal{A}'$. The detailed construction is as follows.

We can assume that $\mathcal{A}_0$ is a complete automaton, otherwise we can simply add a new non-final state q (so-called “dead state”) to Q_0 and all possible transitions with q on the right-hand side to Δ_0.

Let $Q' = 2^{Q_0 \cup \ldots \cup Q_n} \cup \{s_{\text{fin}}\}$ be the powerset of $Q_0 \cup \ldots \cup Q_n$ plus one additional state s_{fin}, which is the only final state of $\mathcal{A}'$, that is $F' = \{s_{\text{fin}}\}$. For $s_1, \ldots, s_k \in Q'$ and k-ary $f \in \Sigma$ we define that $f(s_1, \ldots, s_k) \to s \in \Delta'$ if s is the set

$$\{q \in Q_0 \cup \ldots \cup Q_n \mid \exists q_1 \in s_1 \ldots \exists q_k \in s_k \ f(q_1, \ldots, q_k) \to q \in \Delta_0 \cup \ldots \cup \Delta_n\}.$$

For $s_0, \ldots, s_n \in Q'$ we define that $\#(s_0, \ldots, s_n) \to s_{\text{fin}} \in \Delta'$ if

$$s_0 \cap F_0 = \emptyset, s_1 \cap F_1 \neq \emptyset, \ldots, s_n \cap F_n \neq \emptyset.$$

Finally we define Q as the set of reachable states from Q' (it is well-known that reachability for tree automata can be tested in polynomial time), Δ as the restriction of Δ' to Q, and F as F'.

The correctness of the construction follows immediately from the observation that for $i = 0, \ldots, n$, the automaton

$$\mathcal{A}'_i = \langle \Sigma \cup \{\#\}, Q, \Delta, \{s \in Q \mid s \cap F_i \neq \emptyset\}\rangle$$

recognizes exactly the set $\mathcal{L}(\mathcal{A}_i)$, and $\mathcal{A}'_0$ restricted to Σ is complete. □

Decidability of Problem 1. Let the clause $p_0(t_0) \leftarrow p_1(t_1), \ldots, p_n(t_n)$ and the family of regular sets $T_{p_0}, T_{p_1}, \ldots, T_{p_n}$ be an instance of Problem 1. We did not specify here the formalism in which the sets $T_{p_0}, \ldots, T_{p_n}$ are given, but without loss of generality we can assume that the automata recognizing them are known. The translation from other formalisms like ground set expressions from [1] or regular grammars from [16] is straightforward.

The idea of the proof is to test the emptiness of the intersection of the automaton constructed in Lemma 1 with the set of instances of the term $\#(t_0, \ldots, t_n)$. Due to non-linear occurrences of variables in $\#(t_0, \ldots, t_n)$ this last set is, however, not regular. For our purposes it is enough, however, if we assign the same state of an automaton to each occurrence of the same variable.

Lemma 2. *Let $\mathcal{A} = \langle \Sigma, Q, \Delta, F\rangle$ be a deterministic bottom-up tree automaton without unreachable states, recognizing $\#(T_{\Sigma-\{\#\}} - T_0, T_1, \ldots, T_n)$, as constructed in Lemma 1. Then the set $\bigcup_{i=0}^{n}\{p_i(t) \mid t \in T_{p_i}\}$ is not a model of the clause $p_0(t_0) \leftarrow p_1(t_1), \ldots, p_n(t_n)$ if and only if there exists a mapping $\theta : \mathsf{Var}(\#(t_0, \ldots, t_n)) \to Q$ such that the term $\#(t_0, \ldots, t_n)\theta$ is accepted by the automaton $\mathcal{A}$.*

Proof. The above set is not a model of the clause if and only if there exists a substitution $\sigma : \mathsf{Var}(\#(t_0, \ldots, t_n)) \to T_{\Sigma-\{\#\}}$ such that $t_1\sigma \in T_{p_1}, \ldots, t_n\sigma \in T_{p_n}$ and $t_0\sigma \notin T_{p_0}$. This is equivalent to the existence of such a σ that the automaton $\mathcal{A}$ accepts the term $\#(t_0, \ldots, t_n)\sigma$. Thus it is enough to prove the equivalence of the last condition with the acceptance of $\#(t_0, \ldots, t_n)\theta$ by $\mathcal{A}$.

Now we prove this equivalence. Suppose that $\mathcal{A}$ accepts $\#(t_0, \ldots, t_n)\sigma$ with a run ρ. Note that by the determinism of $\mathcal{A}$, there is only one possible run of $\mathcal{A}$ on $\#(t_0, \ldots, t_n)\sigma$, and for each occurrence of $x\sigma$ the state assigned by ρ is the same, and thus we can speak about states assigned to terms (as opposed to occurrences of terms). Taking $\theta(x) = \rho(\sigma(x))$ we obtain $\rho(\#(t_0, \ldots, t_n)\theta) = \rho(\#(t_0, \ldots, t_n)\sigma) \in F$ and the automaton accepts $\#(t_0, \ldots, t_n)\theta$.

Conversely, suppose there exists θ such that $\#(t_0, \ldots, t_n)\theta$ is accepted by $\mathcal{A}$. Since all states in Q are reachable, there exists a tree t_x accepted by the state $\theta(x)$. Putting $\sigma(x) = t_x$ for all $x \in \mathsf{Var}(\#(t_0, \ldots, t_n))$ we obtain a σ such that $\mathcal{A}$ accepts the term $\#(t_0, \ldots, t_n)\sigma$. □

Theorem 2. *Problem 1 is decidable in DEXPTIME.*

Proof. This is a consequence of the Lemma 2 above: there are $|Q|^{|\mathsf{Var}(\#(t_0, \ldots, t_n))|}$ possible mappings θ; this number is exponential in the size of the input, since $|Q|$ is exponential and $|\mathsf{Var}(\#(t_0, \ldots, t_n))|$ is linear; each such θ can be tested in polynomial time. □

The following corollary is a direct consequence of Theorems 1 and 2, and the DEXPTIME-hardness result from [12].

Corollary 1. *Directional type checking for logic programs wrt. arbitrary regular types is DEXPTIME-complete.*

3.2 Parameterized Complexity of Type Checking

Let us recall the DEXPTIME-hardness proof for the type-checking of logic programs wrt. discriminative types. It is based on a reduction from the emptiness problem for intersection of deterministic top-down tree automata [37]. It is shown that the program consisting of a single clause $p(X, \dots, X)$ is well-typed wrt. $(T_1, \dots, T_n) \rightarrow \emptyset$ if and only if the intersection $T_1 \cap \dots \cap T_n$ is empty. For the hardness proof the sets $T_1, \dots, T_n$ are chosen as discriminative regular sets of trees, whose intersection encodes computation of an alternating Turing machine with polynomialy bounded tape.

What strikes in this construction is that the program used here is very short (it is only one fact) while the types are very large (the encoding of the Turing-machine computation is done in the types, not in the program). The situation in practice looks usually quite opposite: the type-checking problems are usually applied to large programs and rather small types. A natural way to approach such problem is to study its parameterized complexity [18].

A parameterized problem takes as input a pair $\langle x, k \rangle$ where x is a word (in our case the encoding of a logic program and a directional type) and k is a positive integer. Such a problem is called fixed-parameter linear if there exists a function $f : \mathbb{N} \rightarrow \mathbb{N}$ and an algorithm that decides the problem and runs in time $f(k)|x|$.

In the formulation of the problems below, $|\mathcal{T}|$ denotes the size of $\mathcal{T}$. Formally, it is the sum of the lengths of the encodings of the automata recognizing the regular sets occurring in $\mathcal{T}$. Similarly, $|T|$ denotes the size of T (the length of the encoding of the automaton recognizing T).

Problem 2 (Parameterized type-checking).
Instance: a logic program $\mathcal{P}$ and a directional type $\mathcal{T}$
Parameter: $|\mathcal{T}|$
Question: is $\mathcal{P}$ well-typed wrt. $\mathcal{T}$?

Theorem 3. *The parametrized type-checking problem is decidable in time $O(c^{|\mathcal{T}|} \cdot |\mathcal{P}|)$ for some constant c that does not depend on $\mathcal{P}$.*

The proof of this theorem follows directly from Lemma 3

Problem 3 (Parametrized version of Problem 1).
Instance: a clause $p_0(t_0) \leftarrow p_1(t_1), \dots, p_n(t_n)$ and a family of regular sets $T_{p_0}, T_{p_1}, \dots, T_{p_n}$
Parameter: $\sum_{p_i \in \{p_1, \dots, p_n\}} |T_{p_i}|$
Question: is the set $\bigcup_{i=0}^{n} \{p_i(t) \mid t \in T_{p_i}\}$ a model of the clause?

Lemma 3. *Problem 3 is decidable in time $O(c^k m)$ for some constant c that does not depend on m, where k is the parameter and m is the size of the clause.*

Proof. The idea is again to use Lemma 2. We use notations from Lemmas 1 and 2. We traverse the term $\#(t_0, \dots, t_n)$ top-down checking which assumptions have to be made on the value of the run on subterms to make the term accepted by the automaton. These checks will succeed if the assumptions about the run on variables give raise to a function from variables to states of the automaton.

Consider a set of pairs of the form $\{\langle q_1, s_1\rangle, \dots, \langle q_k, s_k\rangle\}$ where q_i is a state of the automaton $\mathcal{A}$ and s_i is a subterm of $\#(t_0, \dots, t_n)$. Intuitively, this set will express the information "if there exists a run ρ of the automaton $\mathcal{A}$ such that $\rho(s_i) = q_i$ then $\mathcal{A}$ accepts $\#(t_0, \dots, t_n)$". We call such a set S *flat* if all terms occurring in S are variables. A flat set S is *inconsistent* if it contains two different pairs $\langle q, x\rangle$ and $\langle q', x\rangle$ with the same variable x and different states q, q'; otherwise it is consistent. A flat and consistent set defines a function assigning states to variables.

Consider a function check assigning a boolean value to such sets of pairs, defined recursively as follows.

$$\mathsf{check}(S) = \begin{cases} \mathit{true}, & \text{if } S \text{ flat and consistent} \\ \mathit{false}, & \text{if } S \text{ flat and inconsistent} \\ \displaystyle\bigvee_{f(q_1,\dots,q_k)\to q\in\Delta} \mathsf{check}(S - \{\langle q, f(s_1,\dots,s_k)\rangle\} \cup \{\langle q_1, s_1\rangle, \dots, \langle q_k, s_k\rangle\}), & \\ & \text{otherwise} \end{cases}$$

We claim that

1. $\mathsf{check}(\{\langle s_{\mathrm{fin}}, \#(t_0, \dots, t_n)\rangle\}) = \mathit{true}$ if and only if there exists a function $\theta : \mathsf{Var}(\#(t_0, \dots, t_n)) \to Q$ such that the term $\#(t_0, \dots, t_n)\theta$ is accepted by the automaton $\mathcal{A}$.
2. the value of $\mathsf{check}(\{\langle s_{\mathrm{fin}}, \#(t_0, \dots, t_n)\rangle\})$ can be computed in time $O(|\#(t_0, \dots, t_n)| \cdot |\mathcal{A}|)$

The first part can be easily proved by induction on the structure of the term $\#(t_0, \dots, t_n)$. The run of the automaton must assign the final state s_{fin} to the term $\#(t_0, \dots, t_n)$, which is expressed by the pair $\langle s_{\mathrm{fin}}, \#(t_0, \dots, t_n)\rangle$; each computation step of the automaton must agree with some transition in Δ, which is expressed by the disjunction over matching transitions in the definition of check; finally the condition that θ is a function is expressed by the consistency of the set S. Note also that by the associativity and commutativity of disjunction, the value of check does not depend on the choice of the non-flat element $\langle q, f(s_1, \dots, s_n)\rangle$ from S.

For the second part, note that for each subterm s of $\#(t_0, \dots, t_n)$ there are at most $|\Delta|$ calls to the function check that correspond to decomposing of the term s. Since there are exactly $|\#(t_0, \dots, t_n)|$ such subterms, the whole work is done in time $O(|\#(t_0, \dots, t_n)| \cdot |\mathcal{A}|)$. □

3.3 Incrementality and Infinite Signature

A literal application of the algorithm presented above might lead to the following problem. Suppose that some program is well-typed and we increment it by adding a new, completely independent, fragment defining a new predicate. The new fragment may contain new function symbols, which did not occur in the original program. Since a signature is a part of the definition of a tree automaton, the old type-check was done with automata over smaller signature, and one could argue that now the type-checking procedure has to be rerun from scratch.

However, it is fairly straightforward to extend tree automata to deal with infinite signature. We can simply consider an infinite signature Σ_{inf} containing Σ, add a new state q_{any} to the automaton and say that the transition relation Δ implicitly contains all transitions of the form $f(\ldots) \to q_{\text{any}}$ for all $f \in \Sigma_{inf}$. Such an automaton has still finite set of states and infinite (but finitely representable) transition relation.

With such an extension of tree automata our algorithm is still correct (a little bit of work has to be done to correctly reason about implicit transitions and reachable states during the determinization step); it is still fixed-parameter linear (when traversing the term $\#(t_0, \ldots, t_n)$ there is no need to look at function symbols that do not occur in this term).

Another problem of the same nature is that numbers (integers or reals) do not form a regular set. In order to extend tree automata to deal with these sets it is enough to treat each number as a constant symbol, add two states int and real and infinitely many implicit transitions $i \to \mathsf{int}$ and $r \to \mathsf{real}$ for all integers i and reals r.

4 Conclusion

We proved the decidability in Dexptime and fixed-parameter linearity of directional-type checking for logic programs wrt. general regular types. This solves a problem that was open since 1994 and improves several earlier partial solutions.

The procedure we presented is optimal from the complexity point of view, it is also incremental. This, together with linear complexity in the size of program gives us a hope that the type system may be usable in practice.

There are some obvious directions for the future work. One is the implementation of the system to see how it behaves in practice. Further, an extension to constraint logic programming, negation etc. would be interesting. The extension to polymorphic types seems not to be very difficult.

Acknowledgments

I thank Andreas Podelski for interesting discussions and the anonymous referees for their comments on the paper.

References

1. A. Aiken and T. K. Lakshman. Directional type checking of logic programs. In B. L. Charlier, editor, *1st International Symposium on Static Analysis*, volume 864 of *Lecture Notes in Computer Science*, pages 43–60, Namur, Belgium, Sept. 1994. Springer Verlag.

2. K. R. Apt. Declarative programming in Prolog. In D. Miller, editor, *Logic Programming - Proceedings of the 1993 International Symposium*, pages 12–35, Vancouver, Canada, 1993. The MIT Press.
3. K. R. Apt. Program verification and Prolog. In E. Börger, editor, *Specification and Validation methods for Programming languages and systems*, pages 55–95. Oxford University Press, 1995.
4. K. R. Apt and S. Etalle. On the unification free Prolog programs. In A. M. Borzyszkowski and S. Sokolowski, editors, *Mathematical Foundations of Computer Science 1993, 18th International Symposium*, volume 711 of *LNCS*, pages 1–19, Gdansk, Poland, 30 Aug.– 3 Sept. 1993. Springer.
5. J. Boye. *Directional Types in Logic Programming.* PhD thesis, Department of Computer and Information Science, Linköping University, 1996.
6. J. Boye and J. Maluszynski. Two aspects of directional types. In L. Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, pages 747–764, Cambridge, June13–18 1995. MIT Press.
7. J. Boye and J. Maluszynski. Directional types and the annotation method. *Journal of Logic Programming*, 33(3):179–220, Dec. 1997.
8. F. Bronsard, T. K. Lakshman, and U. S. Reddy. A framework of directionality for proving termination of logic programs. In K. Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 321–335, Washington, USA, 1992. The MIT Press.
9. A. Caron, J. Coquidé, and M. Dauchet. Encompassment properties and automata with constraints. In C. Kirchner, editor, *5th international conference on Rewriting Techniques and Applications*, LNCS 690, pages 328–342, Montréal, 1993.
10. W. Charatonik. Automata on DAG representations of finite trees. Technical Report MPI-I-1999-2-001, Max-Planck-Institut für Informatik, Mar. 1999. www.mpi-sb.mpg.de/∼witold/papers/dag.ps.
11. W. Charatonik, F. Jacquemard, and A. Podelski. Directional type checking for logic programs is decidable, 1998. Unpublished note.
12. W. Charatonik and A. Podelski. Directional type inference for logic programs. In G. Levi, editor, *Proceedings of the Fifth International Static Analysis Symposium (SAS)*, LNCS 1503, pages 278–294, Pisa, Italy, 1998. Springer-Verlag.
13. W. Charatonik and A. Podelski. Set-based analysis of reactive infinite-state systems. In B. Steffen, editor, *Fourth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1384, pages 358–375, Lisbon, Portugal, March-April 1998. Springer-Verlag.
14. W. Charatonik, A. Podelski, and J.-M. Talbot. Paths vs. trees in set-based program analysis. In *27th Annual ACM Symposium on Principles of Programming Languages*, Jan. 2000.
15. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. www.grappa.univ-lille3.fr/tata.
16. P. Dart and J. Zobel. A regular type language for logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 157–189. MIT Press, 1992.
17. P. Devienne, J.-M. Talbot, and S. Tison. Set-based analysis for logic programming and tree automata. In *Proceedings of the Static Analysis Symposium, SAS'97*, volume 1302 of *LNCS*, pages 127–140. Springer-Verlag, 1997.
18. R. G. Downey and M. Fellows. *Parameterized complexity.* Monographs in computer science. Springer, New York, 1999.

19. T. Frühwirth, E. Shapiro, M. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 300–309, July 1991.
20. J. Gallagher and D. A. de Waal. Regular approximations of logic programs and their uses. Technical Report CSTR-92-06, Department of Computer Science, University of Bristol, 1992.
21. J. Gallagher and D. A. de Waal. Fast and precise regular approximations of logic programs. In P. V. Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming*, pages 599–613, Santa Margherita Ligure, Italy, 1994. The MIT Press.
22. F. Gécseg and M. Steinby. *Tree Automata*. Akademiai Kiado, 1984.
23. G. Gottlob, E. Grädel, and H. Veith. Datalog LITE: Temporal versus deductive reasoning in verification. Technical Report DBAI-TR-98-22, Institut für Informationssysteme, Technische Universität Wien, December 1998.
24. N. Heintze. *Set based program analysis*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1992.
25. N. Heintze and J. Jaffar. A finite presentation theorem for approximating logic programs. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 197–209, January 1990.
26. N. Heintze and J. Jaffar. Semantic types for logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 141–156. MIT Press, 1992.
27. F. Henglein. Type inference with polymorphic recursion. *Transactions on Programming Languages and Systems*, 15(2):253–289, 1993.
28. P. V. Hentenryck, A. Cortesi, and B. L. Charlier. Type analysis of Prolog using type graphs. *Journal of Logic Programming*, 22(3):179–209, Mar. 1995.
29. G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *Journal of Logic Programming*, 13(2-3):205–258, July 1992.
30. P. Mishra. Towards a theory of types in Prolog. In *IEEE International Symposium on Logic Programming*, pages 289–298, 1984.
31. A. Mycroft and R. A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.
32. L. Naish. A declarative view of modes. In *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming*, pages 185–199. MIT Press, September 1996.
33. F. Pfenning, editor. *Types in Logic Programming*. MIT Press, 1992.
34. Y. Ramakrishna, C. Ramakrishnan, I. Ramakrishnan, S. Smolka, T. Swift, and D. Warren. Efficient model checking using tabled resolution. In *Computer Aided Verification (CAV'97)*, LNCS 1254. Springer-Verlag, June 1997.
35. Y. Rouzaud and L. Nguyen-Phuong. Integrating modes and subtypes into a Prolog type-checker. In K. Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 85–97, Washington, USA, 1992. The MIT Press.
36. P. Rychlikowski and T. Truderung. Polymorphic directional types for logic programming. http://www.tcs.uni.wroc.pl/~tomek/dirtypes/, 2000.
37. H. Seidl. Haskell overloading is DEXPTIME-complete. *Information Processing Letters*, 52:57–60, 1994.
38. W. Thomas. *Handbook of Theoretical Computer Science*, volume B, chapter Automata on Infinite Objects, pages 134–191. Elsevier, 1990.
39. E. Yardeni and E. Shapiro. A type system for logic programs. *Journal of Logic Programming*, 10:125–153, 1991.

Formalizing Implementation Strategies for First-Class Continuations[⋆]

Olivier Danvy

BRICS[⋆⋆]
Department of Computer Science, University of Aarhus
Building 540, Ny Munkegade, DK-8000 Aarhus C, Denmark
E-mail: danvy@brics.dk
Home page: http://www.brics.dk/~danvy

Abstract. We present the first formalization of implementation strategies for first-class continuations. The formalization hinges on abstract machines for continuation-passing style (CPS) programs with a special treatment for the current continuation, accounting for the essence of first-class continuations. These abstract machines are proven equivalent to a standard, substitution-based abstract machine. The proof techniques work uniformly for various representations of continuations. As a byproduct, we also present a formal proof of the two folklore theorems that one continuation identifier is enough for second-class continuations and that second-class continuations are stackable.

A large body of work exists on implementing continuations, but it is predominantly empirical and implementation-oriented. In contrast, our formalization abstracts the essence of first-class continuations and provides a uniform setting for specifying and formalizing their representation.

1 Introduction

Be it for coroutines, threads, mobile code, interactive computer games, or computer sessions, one often needs to suspend and to resume a computation. Suspending a computation amounts to saving away its state, and resuming a suspended computation amounts to restoring the saved state. Such saved copies may be ephemeral and restored at most once (e.g., coroutines, threads, and computer sessions that were 'saved to disk'), or they may need to be restored repeatedly (e.g., in a computer game). This functionality is reminiscent of *continuations*, which represent the rest of a computation [22].

In this article, we consider how to implement first-class continuations. A wealth of empirical techniques exist to take a snapshot of control during the execution of a program (call/cc) and to restore this snapshot (throw): SML/NJ, for example, allocates continuations entirely in the heap, reducing call/cc and throw to a matter of swapping pointers [1]; T and Scheme 48 allocate continuations on a stack, copying this stack in the heap and back to account for

[⋆] Extended version available as the technical report BRICS RS-99-51.

[⋆⋆] Basic Research in Computer Science (http://www.brics.dk), Centre of the Danish National Research Foundation.

G. Smolka (Ed.): ESOP/ETAPS 2000, LNCS 1782, pp. 88–103, 2000.

call/cc and throw [16,17];[1] and PC Scheme, Chez Scheme, and Larceny allocate continuations on a segmented stack [2,4,15]. Clinger, Hartheimer, and Ost's recent article [4] provides a comprehensive overview of implementation strategies for first-class continuations and of their issues: ideally, first-class continuations should exert zero overhead for programs that do not use them.

Our goal and non-goal: We formalize implementation strategies for first-class continuations. We do not formalize first-class continuations per se (cf., e.g., Felleisen's PhD thesis [12] or Duba, Harper, and MacQueen's formal account of call/cc in ML [10]).

Our work: We consider abstract machines for continuation-passing style (CPS) programs, focusing on the implementation of continuations. As a stepping stone, we formalize the folklore theorem that one register is enough to implement second-class continuations. We then formalize the three implementation techniques for first-class continuations mentioned above: heap, stack, and segmented stack. The formalization and its proof techniques (structural induction on terms and on derivation trees) are uniform: besides clarifying what it means to implement continuations, be they second-class or first-class, our work provides a platform to state and prove the correctness of each implementation. Also, this platform is not restricted to CPS programs: through Flanagan et al.'s results [13], it is applicable to direct-style programs if one represents control with a stack of evaluation contexts instead of a stack of functions.

1.1 Related Work

The four works most closely related to ours are Clinger, Hartheimer, and Ost's overview of implementation strategies for first-class continuations [4]; Flanagan, Sabry, Duba, and Felleisen's account of compiling with continuations and more specifically, their two first abstract machines [13]; Danvy and Lawall's syntactic characterization of second-class and first-class continuations in CPS programs [8]; and Danvy, Dzafic, and Pfenning's work on the occurrence of continuation parameters in CPS programs [6,9,11].

1.2 Overview

Section 2 presents our source language: the λ-calculus in direct style and in CPS, the CPS transformation, and an abstract machine for CPS programs that will be our reference point here. This standard machine treats continuation identifiers on par with all the other identifiers. The rest of this article focuses on continuation identifiers and how to represent their bindings – i.e., on the essence of how to implement continuations.

[1] This strategy is usually attributed to Drew McDermott in the late 70's [19], but apparently it was already considered in the early '70s at Queen Mary and Westfield College to implement PAL (John C. Reynolds, personal communication, Aarhus, Denmark, fall 1999).

Section 3 addresses second-class continuations. In a CPS program with second-class continuations, continuation identifiers are not only linear (in the sense of Linear Logic), but they also denote a stackable resource, and indeed it is folklore that second-class continuations can be implemented LIFO on a "control stack". We formalize this folklore by characterizing second-class continuations syntactically in a CPS program and by presenting an abstract machine where the bindings of continuation identifiers are represented with a stack. We show this stack machine to be equivalent to the standard one.

Section 4 addresses first-class continuations. In a CPS program with first-class continuations, continuation identifiers do not denote a stackable resource in general. First-class continuations, however, are relatively rare, and thus over the years, "zero-overhead" implementations have been sought [4]: implementations that do support first-class continuations but only tax programs that use them. We consider the traditional strategy of stack-allocating all continuations by default, as if they were all second-class, and of copying this stack in case of first-class continuations. We formalize this empirical strategy with a new abstract machine, which we show to be equivalent to the standard one.

Section 5 outlines how to formalize alternative implementation strategies, such as segmenting the stack and recycling unshared continuations.

2 CPS Programs

We consider closed programs: direct-style (DS) λ-terms with literals. The BNF of DS programs is displayed in Figure 1. Assuming a call-by-value evaluation strategy, the BNF of CPS programs is displayed in Figure 2. CPS programs are prototypically obtained by CPS-transforming DS programs, as defined in Figure 3 [7,20,21].

Figure 4 displays our starting point: a standard abstract machine implementing β-reduction for CPS programs. This machine is a simplified version of another machine studied jointly with Belmina Dzafic and Frank Pfenning [6,9,11]. We use two judgments, indexed by the syntactic categories of CPS terms. The judgment

$$\vdash^{\text{CProg}}_{\text{std}} p \hookrightarrow a$$

is satisfied whenever a CPS program p evaluates to an answer a. The auxiliary judgment

$$\vdash^{\text{CExp}}_{\text{std}} e \hookrightarrow a$$

is satisfied whenever a CPS expression e evaluates to an answer a. The machine starts and stops with the initial continuation k_{init}, which is a distinguished fresh continuation identifier. Answers can be either the trivial expressions ℓ or $\lambda x.\lambda k.e$, or the error token.

For expository simplicity, our standard machine uses substitutions to implement variable bindings. Alternatively and equivalently, it could use an environment and represent functional values as closures [18]. And indeed Flanagan et al. present a similar standard abstract machine which uses an environment [13, Figure 4].

$p \in$ DProg	— DS programs	$p ::= e$
$e \in$ DExp	— DS expressions	$e ::= e_0\, e_1 \mid t$
$t \in$ DTriv	— DS trivial expressions	$t ::= \ell \mid x \mid \lambda x.e$
$\ell \in$ Lit	— literals	
$x \in$ Ide	— identifiers	

Fig. 1. BNF of DS programs

$p \in$ CProg	— CPS programs	$p ::= \lambda k.e$
$e \in$ CExp	— CPS (serious) expressions	$e ::= t_0\, t_1\, c \mid c\, t$
$t \in$ CTriv	— CPS trivial expressions	$t ::= \ell \mid x \mid v \mid \lambda x.\lambda k.e$
$c \in$ Cont	— continuations	$c ::= \lambda v.e \mid k$
$\ell \in$ Lit	— literals	
$x \in$ Ide	— source identifiers	
$k \in$ IdeC	— fresh continuation identifiers	
$v \in$ IdeV	— fresh parameters of continuations	
$a \in$ Answer	— CPS answers	$a ::= \ell \mid \lambda x.\lambda k.e \mid \text{error}$

Fig. 2. BNF of CPS programs

$$[\![e]\!]^{\text{DProg}}_{\text{cps}} = \lambda k.[\![e]\!]^{\text{DExp}}_{\text{cps}}\, k \qquad \text{– where } k \text{ is fresh}$$

$$[\![e_0\, e_1]\!]^{\text{DExp}}_{\text{cps}} c = [\![e_0]\!]^{\text{DExp}}_{\text{cps}} \lambda v_0.[\![e_1]\!]^{\text{DExp}}_{\text{cps}} \lambda v_1.v_0\, v_1\, c \qquad \text{– where } v_0 \text{ and } v_1 \text{ are fresh}$$

$$[\![t]\!]^{\text{DExp}}_{\text{cps}} c = c\, [\![t]\!]^{\text{DTriv}}_{\text{cps}}$$

$$[\![\ell]\!]^{\text{DTriv}}_{\text{cps}} = \ell$$

$$[\![x]\!]^{\text{DTriv}}_{\text{cps}} = x$$

$$[\![\lambda x.e]\!]^{\text{DTriv}}_{\text{cps}} = \lambda x.\lambda k.[\![e]\!]^{\text{DExp}}_{\text{cps}}\, k \qquad \text{– where } k \text{ is fresh}$$

Fig. 3. The left-to-right, call-by-value CPS transformation

$$\frac{\vdash^{\text{CExp}}_{\text{std}} e[k_{\text{init}}/k] \hookrightarrow a}{\vdash^{\text{CProg}}_{\text{std}} \lambda k.e \hookrightarrow a}$$

$$\frac{}{\vdash^{\text{CExp}}_{\text{std}} \ell\, t\, c \hookrightarrow \text{error}} \qquad \frac{\vdash^{\text{CExp}}_{\text{std}} e[t/x,\, c/k] \hookrightarrow a}{\vdash^{\text{CExp}}_{\text{std}} (\lambda x.\lambda k.e)\, t\, c \hookrightarrow a}$$

$$\frac{\vdash^{\text{CExp}}_{\text{std}} e[t/v] \hookrightarrow a}{\vdash^{\text{CExp}}_{\text{std}} (\lambda v.e)\, t \hookrightarrow a} \qquad \frac{}{\vdash^{\text{CExp}}_{\text{std}} k_{\text{init}}\, t \hookrightarrow t}$$

Fig. 4. Standard machine for CPS programs

3 A Stack Machine for CPS Programs with Second-Class Continuations

As a stepping stone, this section formalizes the folklore theorem that in the absence of first-class continuations, one continuation identifier is enough, i.e., in Figure 2, IdeC can be defined as a singleton set. To this end, we prove that in the output of the CPS transformation, only one continuation identifier is indeed enough. We also prove that this property is closed under arbitrary β-reduction. We then rephrase the BNF of CPS programs with IdeC as a singleton set (Section 3.1). In the new BNF, only CPS programs with second-class continuations can be expressed. We present a stack machine for these CPS programs and we prove it equivalent to the standard machine of Figure 4 (Section 3.2). Flanagan et al. present a similar abstract machine [13, Figure 5], but without relating it formally to their standard abstract machine.

3.1 One Continuation Identifier is Enough

Each expression in a DS program occurs in one evaluation context. Correspondingly, each expression in a CPS program has one continuation. We formalize this observation in terms of continuation identifiers with the judgment defined in Figure 5, where $\mathrm{FC}(t)$ yields the set of continuation identifiers occurring free in t.

Definition 1 (Second-class position, second-class continuations). *In a continuation abstraction $\lambda k.e$, we say that k occurs in* second-class position *and denotes a* second-class continuation *whenever the judgment $k \models^{\mathrm{CExp}}_{\mathit{2cc}} e$ is satisfied.*

Below, we prove that actually, in the output of the CPS transformation, *all* continuation identifiers denote second-class continuations. In Figure 6, we thus generalize our judgment to a whole CPS program.

Definition 2 (2Cont-validity). *We say that a CPS program p is* 2Cont-valid *whenever the judgment $\models^{\mathrm{CProg}}_{\mathit{2cc*}} p$ is satisfied. Informally, $\models^{\mathrm{CProg}}_{\mathit{2cc*}} p$ holds if and only if all continuation abstractions $\lambda k.e$ occurring in p satisfy $k \models^{\mathrm{CExp}}_{\mathit{2cc}} e$.*

Lemma 1 (The CPS transformation yields 2Cont-valid programs). *For any $p \in \mathrm{DProg}$, $\models^{\mathrm{CProg}}_{\mathit{2cc*}} [\![p]\!]^{\mathrm{DProg}}_{\mathrm{cps}}$.*

Proof. A straightforward induction over DS programs. □

$$\frac{k \notin \mathrm{FC}(t_0) \qquad k \notin \mathrm{FC}(t_1) \qquad k \models^{\mathrm{Cont}}_{\mathrm{2cc}} c}{k \models^{\mathrm{CExp}}_{\mathrm{2cc}} t_0\, t_1\, c} \qquad \frac{k \models^{\mathrm{Cont}}_{\mathrm{2cc}} c \qquad k \notin \mathrm{FC}(t)}{k \models^{\mathrm{CExp}}_{\mathrm{2cc}} c\, t}$$

$$\frac{k \models^{\mathrm{CExp}}_{\mathrm{2cc}} e}{k \models^{\mathrm{Cont}}_{\mathrm{2cc}} \lambda v.e} \qquad \frac{}{k \models^{\mathrm{Cont}}_{\mathrm{2cc}} k}$$

Fig. 5. Characterization of a second-class continuation abstraction $\lambda k.e$

$$\frac{k \models^{\mathrm{CExp}}_{\mathrm{2cc*}} e}{\models^{\mathrm{CProg}}_{\mathrm{2cc*}} \lambda k.e}$$

$$\frac{\models^{\mathrm{CTriv}}_{\mathrm{2cc*}} t_0 \qquad \models^{\mathrm{CTriv}}_{\mathrm{2cc*}} t_1 \qquad k \models^{\mathrm{Cont}}_{\mathrm{2cc*}} c}{k \models^{\mathrm{CExp}}_{\mathrm{2cc*}} t_0\, t_1\, c} \qquad \frac{k \models^{\mathrm{Cont}}_{\mathrm{2cc*}} c \qquad \models^{\mathrm{CTriv}}_{\mathrm{2cc*}} t}{k \models^{\mathrm{CExp}}_{\mathrm{2cc*}} c\, t}$$

$$\frac{}{\models^{\mathrm{CTriv}}_{\mathrm{2cc*}} \ell} \qquad \frac{}{\models^{\mathrm{CTriv}}_{\mathrm{2cc*}} x} \qquad \frac{}{\models^{\mathrm{CTriv}}_{\mathrm{2cc*}} v} \qquad \frac{k \models^{\mathrm{CExp}}_{\mathrm{2cc*}} e}{\models^{\mathrm{CTriv}}_{\mathrm{2cc*}} \lambda x.\lambda k.e}$$

$$\frac{k \models^{\mathrm{CExp}}_{\mathrm{2cc*}} e}{k \models^{\mathrm{Cont}}_{\mathrm{2cc*}} \lambda v.e} \qquad \frac{}{k \models^{\mathrm{Cont}}_{\mathrm{2cc*}} k}$$

Fig. 6. Characterization of a CPS program with second-class continuations

Furthermore, 2Cont-validity is closed under β-reduction, which means that it is preserved by regular evaluation as well as by the arbitrary simplifications of a CPS compiler [21]. The corresponding formal statement and its proof are straightforward and omitted here: we rely on them in the proof of Theorem 1.

Therefore each use of each continuation identifier k is uniquely determined, capturing the fact that in the BNF of 2Cont-valid CPS programs, one continuation identifier is enough. To emphasize this fact, let us specialize the BNF of Figure 2 by defining IdeC as the singleton set $\{\star\}$, yielding the BNF of 2CPS programs displayed in Figure 7.

$p \in$ 2CProg	— 2CPS programs	$p ::= \lambda\star.e$
$e \in$ 2CExp	— 2CPS (serious) expressions	$e ::= t_0\, t_1\, c \mid c\, t$
$t \in$ 2CTriv	— 2CPS trivial expressions	$t ::= \ell \mid x \mid v \mid \lambda x.\lambda\star.e$
$c \in$ 2Cont	— continuations	$c ::= \lambda v.e \mid \star$
$\ell \in$ Lit	— literals	
$x \in$ Ide	— source identifiers	
$\star \in$ Token	— single continuation identifier	
$v \in$ IdeV	— fresh parameters of continuations	
$a \in$ 2Answer	— 2CPS answers	$a ::= \ell \mid \lambda x.\lambda\star.e \mid$ error

Fig. 7. BNF of 2CPS programs

Let $[\![\cdot]\!]^{\mathrm{CProg}}_{\mathrm{strip}}$ denote the straightforward homomorphic mapping from a 2Cont-valid CPS program to a 2CPS program and $[\![\cdot]\!]^{\mathrm{2CProg}}_{\mathrm{name}}$ denote its inverse, such that $\forall p \in \mathrm{CProg}, [\![[\![p]\!]^{\mathrm{CProg}}_{\mathrm{strip}}]\!]^{\mathrm{2CProg}}_{\mathrm{name}} \equiv_\alpha p$ whenever the judgment $\models^{\mathrm{CProg}}_{\mathrm{2cc*}} p$ is satisfied, and $\forall p' \in \mathrm{2CProg}, [\![[\![p]\!]^{\mathrm{2CProg}}_{\mathrm{name}}]\!]^{\mathrm{CProg}}_{\mathrm{strip}} = p'$. These two translations are generalized in Section 4 and thus we omit their definition here.

3.2 A Stack Machine for 2CPS Programs

Figure 8 displays a stack-based abstract machine for 2CPS programs. We obtained it from the standard machine of Section 2, page 91, by implementing the bindings of continuation identifiers with a global "control stack" φ.

$\varphi \in$ 2CStack — control stacks $\qquad\qquad \varphi ::= \bullet \mid \varphi,\, \lambda v.e$

The machine starts and stops with an empty control stack $\bullet$. When a function is applied, its continuation is pushed on φ. When a continuation is needed, it is popped from φ. If φ is empty, the intermediate result sent to the continuation is the final answer. We distinguish tail calls (i.e., function calls where the continuation is $\star$) by not pushing anything on φ, thereby achieving proper tail recursion.

$$\frac{\bullet \vdash^{\text{2CExp}}_{\text{2cc}} e \hookrightarrow a}{\vdash^{\text{2CProg}}_{\text{2cc}} \lambda\star.e \hookrightarrow a} \qquad \frac{}{\varphi \vdash^{\text{2CExp}}_{\text{2cc}} \ell\, t\, c \hookrightarrow \text{error}}$$

$$\frac{\varphi \vdash^{\text{2CExp}}_{\text{2cc}} e[t/x] \hookrightarrow a}{\varphi \vdash^{\text{2CExp}}_{\text{2cc}} (\lambda x.\lambda\star.e)\, t\, \star \hookrightarrow a} \qquad \frac{\varphi,\, \lambda v.e' \vdash^{\text{2CExp}}_{\text{2cc}} e[t/x] \hookrightarrow a}{\varphi \vdash^{\text{2CExp}}_{\text{2cc}} (\lambda x.\lambda\star.e)\, t\, \lambda v.e' \hookrightarrow a}$$

$$\frac{\varphi \vdash^{\text{2CExp}}_{\text{2cc}} e[t/v] \hookrightarrow a}{\varphi \vdash^{\text{2CExp}}_{\text{2cc}} (\lambda v.e)\, t \hookrightarrow a} \qquad \frac{}{\bullet \vdash^{\text{2CExp}}_{\text{2cc}} \star\, t \hookrightarrow t} \qquad \frac{\varphi \vdash^{\text{2CExp}}_{\text{2cc}} e[t/v] \hookrightarrow a}{\varphi,\, \lambda v.e \vdash^{\text{2CExp}}_{\text{2cc}} \star\, t \hookrightarrow a}$$

Fig. 8. Stack machine for 2CPS programs

N.B. The machine does not substitute continuations for continuation identifiers, and therefore one might be surprised by the rule handling the redex $(\lambda v.e)\, t$. Such redexes, however, can occur in the source program.

Formally, the judgment

$$\vdash^{\text{2CProg}}_{\text{2cc}} p \hookrightarrow a$$

is satisfied whenever a CPS program $p \in$ 2CProg evaluates to an answer $a \in$ 2Answer. The auxiliary judgment

$$\varphi \vdash^{\text{2CExp}}_{\text{2cc}} e \hookrightarrow a$$

is satisfied whenever an expression $e \in$ 2CExp evaluates to an answer a, given a control stack $\varphi \in$ 2CStack.

We prove the equivalence between the stack machine and the standard machine by showing that the computations for each abstract machine (represented by derivations) are in bijective correspondence. To this end, we define a "control-stack substitution" over the state of the stack machine (i.e., expression under evaluation and current control stack) to obtain the state of the standard machine (i.e., expression under evaluation). We define control-stack substitution inductively over 2CPS expressions and continuations.

Definition 3 (Control-stack substitution for 2CPS programs). *Given a stack φ of* 2Cont *continuations, the stack substitution of any $e \in$* 2CExp *(resp. $c \in$* 2Cont*), noted $e\{\varphi\}_2$ (resp. $c\{\varphi\}_2$), yields a* CExp *expression (resp. a* Cont *continuation) and is defined as follows.*

$$(t_0\, t_1\, c)\{\varphi\}_2 = [\![t_0]\!]^{\text{2CTriv}}_{\text{name}}\, [\![t_1]\!]^{\text{2CTriv}}_{\text{name}}\, (c\{\varphi\}_2) \qquad (\lambda v.e)\{\varphi\}_2 = \lambda v.(e\{\varphi\}_2)$$
$$(c\, t)\{\varphi\}_2 = (c\{\varphi\}_2)\, [\![t]\!]^{\text{2CTriv}}_{\text{name}} \qquad \star\{\bullet\}_2 = k_{init}$$
$$\star\{\varphi,\, \lambda v.e\}_2 = \lambda v.(e\{\varphi\}_2)$$

Stack substitution is our key tool for mapping a state of the stack machine into a state of the standard machine. It yields CExp expressions and Cont continuations that have one free continuation identifier: k_{init}.

Lemma 2 (2Cont-validity of stack-substituted expressions and continuations).

1. *For any* $e \in$ 2CExp *and for any stack of* 2Cont *continuations* φ*, the judgment* $k_{init} \models^{\text{CExp}}_{2cc*} e\{\varphi\}_2$ *is satisfied.*
2. *For any* $c \in$ 2Cont *and for any stack of* 2Cont *continuations* φ*, the judgment* $k_{init} \models^{\text{Cont}}_{2cc*} c\{\varphi\}_2$ *is satisfied.*

Proof. By mutual induction on the structure of e and c. □

Lemma 3 (Control-stack substitution for 2CPS programs).

1. *For any* $e' \in$ CExp *satisfying* $k \models^{\text{CExp}}_{2cc*} e'$ *for some* k *and for any stack of* 2Cont *continuations* φ*,* $[\![e']\!]^{\text{CExp}}_{\text{strip}}\{\varphi\}_2 = e'[\star\{\varphi\}_2/k]$.
2. *For any* $e \in$ 2CExp*, for any* $t' \in$ CTriv *satisfying* $\models^{\text{CTriv}}_{2cc*} t'$*, for any identifier* i *in* Ide *or in* IdeV*, and for any stack of* 2Cont *continuations* φ*,* $e[[\![t']\!]^{\text{CTriv}}_{\text{strip}}/i]\{\varphi\}_2 = e\{\varphi\}_2[t'/i]$.

Theorem 1 (Simulation). *The stack machine of Figure 8 and the standard machine are equivalent:*

1. *For any 2Cont-valid CPS program* p*,*
   $\vdash^{\text{CProg}}_{std} p \hookrightarrow a$ *if and only if* $\vdash^{\text{2CProg}}_{2cc} [\![p]\!]^{\text{CProg}}_{\text{strip}} \hookrightarrow [\![a]\!]^{\text{Answer}}_{\text{strip}}$.
2. *For any CPS expression* e *satisfying* $k \models^{\text{CExp}}_{2cc*} e$ *for some* k *and for any stack of* 2Cont *continuations* φ*,*
   $\vdash^{\text{CExp}}_{std} [\![e]\!]^{\text{CExp}}_{\text{strip}}\{\varphi\}_2 \hookrightarrow a$ *if and only if* $\varphi \vdash^{\text{2CExp}}_{2cc} [\![e]\!]^{\text{CExp}}_{\text{strip}} \hookrightarrow [\![a]\!]^{\text{Answer}}_{\text{strip}}$.

Proof. The theorem follows in each direction by an induction over the structure of the derivations, using Lemma 3. Let us show the case of tail calls in one direction.

$$\text{Case } \mathcal{E} = \frac{\overset{\mathcal{E}_1}{\varphi \vdash^{\text{2CExp}}_{\text{2cc}} e[t/x] \hookrightarrow [\![a]\!]^{\text{Answer}}_{\text{strip}}}}{\varphi \vdash^{\text{2CExp}}_{\text{2cc}} (\lambda x.\lambda\star.e)\, t\, \star \hookrightarrow [\![a]\!]^{\text{Answer}}_{\text{strip}}},$$

where $\mathcal{E}_1$ names the derivation ending in $\varphi \vdash^{\text{2CExp}}_{\text{2cc}} e[t/x] \hookrightarrow [\![a]\!]^{\text{Answer}}_{\text{strip}}$.

By applying the induction hypothesis to $\mathcal{E}_1$, we obtain a derivation

$$\overset{\mathcal{E}_1'}{\vdash^{\text{CExp}}_{\text{std}} e[t/x]\{\varphi\}_2 \hookrightarrow a}$$

Since $e[t/x]$ is a 2CPS expression, there exists a CPS expression e' satisfying $k \models^{\text{CExp}}_{\text{2cc*}} e'$ for some k and there exists a CPS trivial expression t' satisfying $\models^{\text{CTriv}}_{\text{2cc*}} t'$ such that $e = [\![e']\!]^{\text{CExp}}_{\text{strip}}$ and $t = [\![t']\!]^{\text{CTriv}}_{\text{strip}}$.

By Lemma 3,

$$\begin{aligned}
[\![e']\!]^{\text{CExp}}_{\text{strip}}[[\![t']\!]^{\text{CExp}}_{\text{strip}}/x]\{\varphi\}_2 &= [\![e']\!]^{\text{CExp}}_{\text{strip}}\{\varphi\}_2[t'/x] \\
&= e'[\star\{\varphi\}_2/k][t'/x] \\
&= e'[t'/x,\, \star\{\varphi\}_2/k] && \text{– because } t' \text{ has no free } k \\
& && \text{and } \varphi \text{ has no free } x.
\end{aligned}$$

By inference,

$$\frac{\vdash^{\text{CExp}}_{\text{std}} e'[t'/x,\, \star\{\varphi\}_2/k] \hookrightarrow a}{\vdash^{\text{CExp}}_{\text{std}} (\lambda x.\lambda k.e')\, t'\, (\star\{\varphi\}_2) \hookrightarrow a}$$

Now by definition of stack substitution,

$$(\lambda x.\lambda k.e')\, t'\, (\star\{\varphi\}_2) = [\![(\lambda x.\lambda k.e)\, t\, k']\!]^{\text{CExp}}_{\text{strip}}\{\varphi\}_2, \qquad \text{– for some } k'.$$

In other words, there exists a derivation

$$\frac{\begin{array}{c}\mathcal{E}'_1 \\ \vdash^{\text{CExp}}_{\text{std}} [\![e[t/x]]\!]^{\text{CExp}}_{\text{strip}}\{\varphi\}_2 \hookrightarrow a\end{array}}{\vdash^{\text{CExp}}_{\text{std}} [\![(\lambda x.\lambda k.e)\, t\, k']\!]^{\text{CExp}}_{\text{strip}}\{\varphi\}_2 \hookrightarrow a}$$

which is what we wanted to show. □

3.3 Summary and Conclusion

As a stepping stone towards Section 4, we have formalized and proven two folklore theorems: (1) for CPS programs with second-class continuations, one identifier is enough; and (2) the bindings of continuation identifiers can be implemented with a stack for CPS programs with second-class continuations. To this end, we have considered a simplified abstract machine and taken the same conceptual steps as in our earlier joint work with Dzafic and Pfenning [6,9,11]. This earlier work is formalized in Elf, whereas the present work is not (yet). The rest of this article reports an independent foray. In the next section, we adapt the stack machine to CPS programs with first-class continuations, thereby formalizing an empirical implementation strategy for first-class continuations.

4 A Stack Machine for CPS Programs with First-Class Continuations

First-class continuations occur because of call/cc. The call-by-value CPS transformation of call/cc reads as follows.

$$[\![\text{call/cc}\, e]\!]^{\text{DExp}}_{\text{cps}} c = [\![e]\!]^{\text{DExp}}_{\text{cps}} \lambda f.f\, (\lambda x.\lambda k.c\, x)\, c \quad \text{– where } f,\ x, \text{ and } k \text{ are fresh.}$$

On the right-hand-side of this definitional equation, c occurs twice: once as a regular, second-class continuation, and once more, in $\lambda x.\lambda k.c\, x$. In that term, k is declared but not used – c is used instead and denotes a first-class continuation.

Such CPS programs do not satisfy the judgments of Figures 5 and 6. And indeed, Danvy and Lawall observed that in a CPS program, first-class continuations can be detected through continuation identifiers occurring “out of turn”, so to speak [8].

Because it makes no assumptions on the binding discipline of continuation identifiers, the standard machine of Section 2, page 91, properly handles CPS programs with first-class continuations. First-class continuations, however, disqualify the stack machine of Section 3, page 94.

The goal of this section is to develop a stack machine for CPS programs with first-class continuations. To this end, we formalize what it means for a continuation identifier to occur in first-class position. We also prove that arbitrary β-reduction never promotes a continuation identifier occurring in second-class position into one occurring in first-class position. We then rephrase the BNF of CPS programs to single out continuation identifiers occurring in first-class position and their declaration. And similarly to Section 3, we tag with “$\star$” all the declarations of continuation identifiers occurring in second-class position or not occurring at all, and all second-class positions of continuation identifiers (Section 4.1). We then present a stack machine for these 1CPS programs that copies the stack when first-class continuation abstractions are invoked. We prove it equivalent to the standard machine of Figure 4 (Section 4.2).

4.1 One Continuation Identifier is Not Enough

Following Danvy and Lawall [8], we now say that a continuation identifier occurs in first-class position whenever it occurs elsewhere than in second-class position, which is syntactically easy to detect. We formalize first-class occurrences with the judgment displayed in Figure 9.

$$\frac{k \in \mathrm{FC}(t_0)}{k \models^{\mathrm{CExp}}_{\mathrm{1cc}} t_0\, t_1\, c} \qquad \frac{k \in \mathrm{FC}(t_1)}{k \models^{\mathrm{CExp}}_{\mathrm{1cc}} t_0\, t_1\, c} \qquad \frac{k \models^{\mathrm{Cont}}_{\mathrm{1cc}} c}{k \models^{\mathrm{CExp}}_{\mathrm{1cc}} t_0\, t_1\, c}$$

$$\frac{k \models^{\mathrm{Cont}}_{\mathrm{1cc}} c}{k \models^{\mathrm{CExp}}_{\mathrm{1cc}} c\, t} \qquad \frac{k \in \mathrm{FC}(t)}{k \models^{\mathrm{CExp}}_{\mathrm{1cc}} c\, t}$$

$$\frac{k \models^{\mathrm{CExp}}_{\mathrm{1cc}} e}{k \models^{\mathrm{Cont}}_{\mathrm{1cc}} \lambda v.e}$$

Fig. 9. Characterization of a first-class continuation abstraction $\lambda k.e$

Definition 4 (First-class position, first-class continuations). *In a continuation abstraction $\lambda k.e$, we say that k occurs in* first-class position *and denotes a* first-class continuation *whenever the judgment $k \models^{\mathrm{CExp}}_{1cc} e$ is satisfied.*

N.B. For any continuation abstraction $\lambda k.e$, at most one of $k \models^{\mathrm{CExp}}_{\mathrm{1cc}} e$ and $k \models^{\mathrm{CExp}}_{\mathrm{2cc}} e$ is satisfied.

In Section 3, we stated that 2Cont-validity is closed under β-reduction. Similarly here, β-reduction may demote a first-class continuation identifier into a second-class one, but it can never promote a second-class continuation identifier into a first-class one. The corresponding formal statement and its proof are straightforward and omitted here: we rely on them in the proof of Theorem 2.

For example, in

$$\lambda k.(\lambda x.\lambda k'.k\,x)\,\ell\,k$$

k occurs in first-class position. However, β-reducing this term yields

$$\lambda k.k\,\ell$$

where k occurs in second-class position.

In Section 3, we capitalized on the fact that each second-class position was uniquely determined. Here, we still capitalize on this fact by only singling out continuation identifiers in first-class position.[2]

Introduction: For all continuation abstractions $\lambda k.e$ satisfying $k \models^{\text{CExp}}_{\text{1cc}} e$, we tag the declaration of k with λ^1 and we keep the name k. Otherwise, we replace it with $\star$.

Elimination: When a continuation identifier occurs, if it is the latest one declared, we replace it with $\star$; otherwise, we keep its name.

The resulting BNF for 1CPS programs is displayed in Figure 10. The back and forth translation functions are displayed in Figures 11 and 12. They generalize their counterpart in Section 3.

Lemma 4 (Inverseness of stripping and naming).
$\forall p \in \text{CProg}, [\![[\![p]\!]^{\text{CProg}}_{\text{strip}}]\!]^{\text{1CProg}}_{\text{name}} \equiv_\alpha p$ *and* $\forall p' \in \text{1CProg}, [\![[\![p']\!]^{\text{1CProg}}_{\text{name}}]\!]^{\text{CProg}}_{\text{strip}} = p'$.

4.2 A Stack Machine for CPS Programs with First-Class Continuations

We handle first-class continuations by extending the formalization of Section 3 with a new syntactic form:

$$c \in \text{1Cont} \quad \text{— continuations} \qquad\qquad c ::= \lambda v.e \mid \star \mid k \mid \text{swap}\,\varphi$$

The new form swap φ makes it possible to represent a copy of the control stack φ. It requires us to extend control-stack substitution as follows.

Definition 5 (Control-stack substitution for 1CPS programs). *Given a stack φ of* 1Cont *continuations, The stack substitution of any $e \in$* 1CExp *(resp. $c \in$* 1Cont*), noted $e\{\varphi\}_1$ (resp. $c\{\varphi\}_1$), yields a* CExp *expression (resp. a* Cont *continuation) and is defined as follows.*

$$
\begin{aligned}
(t_0\,t_1\,c)\{\varphi\}_1 &= ([\![t_0]\!]^{\text{1CTriv}}_{\text{name}}\,[\![t_1]\!]^{\text{1CTriv}}_{\text{name}})\,(c\{\varphi\}_1)\\
(c\,t)\{\varphi\}_1 &= (c\{\varphi\}_1)\,[\![t]\!]^{\text{1CTriv}}_{\text{name}}
\end{aligned}
\qquad
\begin{aligned}
(\lambda v.e)\{\varphi\}_1 &= \lambda v.(e\{\varphi\}_1)\\
\star\{\bullet\}_1 &= k_{init}\\
\star\{\varphi,\,\lambda v.e\}_1 &= \lambda v.(e\{\varphi\}_1)\\
k\{\varphi\}_1 &= k\\
(\text{swap}\,\varphi')\{\varphi\}_1 &= \star\{\varphi'\}_1
\end{aligned}
$$

[2] Andrzej Filinski suggested this concise notation (personal communication, Aarhus, Denmark, summer 1999).

$p \in$ 1CProg	— 1CPS programs	$p ::= \lambda\star.e \mid \lambda^1 k.e$
$e \in$ 1CExp	— 1CPS (serious) expressions	$e ::= t_0\, t_1\, c \mid c\, t$
$t \in$ 1CTriv	— 1CPS trivial expressions	$t ::= \ell \mid x \mid v \mid \lambda x.\lambda\star.e \mid \lambda x.\lambda^1 k.e$
$c \in$ 1Cont	— continuations	$c ::= \lambda v.e \mid \star \mid k$
$\ell \in$ Lit	— literals	
$x \in$ Ide	— source identifiers	
$k \in$ IdeC	— fresh continuation identifiers	
$\star \in$ Token	— single continuation identifier	
$v \in$ IdeV	— fresh parameters of continuations	
$a \in$ 1Answer	— 1CPS answers	$a ::= \ell \mid \lambda x.\lambda\star.e \mid \lambda x.\lambda^1 k.e \mid \text{error}$

Fig. 10. BNF of 1CPS programs

$$\llbracket \lambda k.e \rrbracket^{\text{CProg}}_{\text{strip}} = \begin{cases} \lambda^1 k.\llbracket e \rrbracket^{\text{CExp}}_{\text{strip}} k & \text{if } k \models^{\text{CExp}}_{\text{1cc}} e \\ \lambda\star.\llbracket e \rrbracket^{\text{CExp}}_{\text{strip}} k & \text{otherwise} \end{cases}$$

$$\begin{aligned} \llbracket t_0\, t_1\, c \rrbracket^{\text{CExp}}_{\text{strip}} k &= \llbracket t_0 \rrbracket^{\text{CTriv}}_{\text{strip}}\, \llbracket t_1 \rrbracket^{\text{CTriv}}_{\text{strip}}\, (\llbracket c \rrbracket^{\text{Cont}}_{\text{strip}} k) \\ \llbracket c\, t \rrbracket^{\text{CExp}}_{\text{strip}} k &= (\llbracket c \rrbracket^{\text{Cont}}_{\text{strip}} k)\, \llbracket t \rrbracket^{\text{CTriv}}_{\text{strip}} \end{aligned} \qquad \begin{aligned} \llbracket \ell \rrbracket^{\text{CTriv}}_{\text{strip}} &= \ell \\ \llbracket x \rrbracket^{\text{CTriv}}_{\text{strip}} &= x \\ \llbracket v \rrbracket^{\text{CTriv}}_{\text{strip}} &= v \end{aligned}$$

$$\llbracket \lambda x.\lambda k.e \rrbracket^{\text{CTriv}}_{\text{strip}} = \begin{cases} \lambda x.\lambda^1 k.\llbracket e \rrbracket^{\text{CExp}}_{\text{strip}} k & \text{if } k \models^{\text{CExp}}_{\text{1cc}} e \\ \lambda x.\lambda\star.\llbracket e \rrbracket^{\text{CExp}}_{\text{strip}} k & \text{otherwise} \end{cases}$$

$$\llbracket \lambda v.e \rrbracket^{\text{Cont}}_{\text{strip}} k = \lambda v.\llbracket e \rrbracket^{\text{CExp}}_{\text{strip}} k$$

$$\llbracket k' \rrbracket^{\text{Cont}}_{\text{strip}} k = \begin{cases} \star & \text{if } k = k' \\ k' & \text{otherwise} \end{cases}$$

Fig. 11. Translation from CPS to 1CPS – stripping continuation identifiers

$$\begin{aligned} \llbracket \lambda\star.e \rrbracket^{\text{1CProg}}_{\text{name}} &= \lambda k.\llbracket e \rrbracket^{\text{1CExp}}_{\text{name}} k && \text{– where } k \text{ is fresh} \\ \llbracket \lambda^1 k.e \rrbracket^{\text{1CProg}}_{\text{name}} &= \lambda k.\llbracket e \rrbracket^{\text{1CExp}}_{\text{name}} k \end{aligned}$$

$$\begin{aligned} \llbracket t_0\, t_1\, c \rrbracket^{\text{1CExp}}_{\text{name}} k &= \llbracket t_0 \rrbracket^{\text{1CTriv}}_{\text{name}}\, \llbracket t_1 \rrbracket^{\text{1CTriv}}_{\text{name}}\, (\llbracket c \rrbracket^{\text{1Cont}}_{\text{name}} k) \\ \llbracket c\, t \rrbracket^{\text{1CExp}}_{\text{name}} k &= (\llbracket c \rrbracket^{\text{1Cont}}_{\text{name}} k)\, \llbracket t \rrbracket^{\text{1CTriv}}_{\text{name}} \end{aligned} \qquad \begin{aligned} \llbracket \ell \rrbracket^{\text{1CTriv}}_{\text{name}} &= \ell \\ \llbracket x \rrbracket^{\text{1CTriv}}_{\text{name}} &= x \\ \llbracket v \rrbracket^{\text{1CTriv}}_{\text{name}} &= v \end{aligned}$$

$$\begin{aligned} \llbracket \lambda x.\lambda\star.e \rrbracket^{\text{1CTriv}}_{\text{name}} &= \lambda x.\lambda k.\llbracket e \rrbracket^{\text{1CExp}}_{\text{name}} k && \text{– where } k \text{ is fresh} \\ \llbracket \lambda^1 x.\lambda k.e \rrbracket^{\text{1CTriv}}_{\text{name}} &= \lambda x.\lambda k.\llbracket e \rrbracket^{\text{1CExp}}_{\text{name}} k \end{aligned}$$

$$\begin{aligned} \llbracket \lambda v.e \rrbracket^{\text{1Cont}}_{\text{name}} k &= \lambda v.\llbracket e \rrbracket^{\text{1CExp}}_{\text{name}} k \\ \llbracket \star \rrbracket^{\text{1Cont}}_{\text{name}} k &= k \\ \llbracket k' \rrbracket^{\text{1Cont}}_{\text{name}} k &= k' \end{aligned}$$

$$\begin{aligned} \llbracket \ell \rrbracket^{\text{1Answer}}_{\text{name}} &= \ell \\ \llbracket \lambda x.\lambda\star.e \rrbracket^{\text{1Answer}}_{\text{name}} &= \lambda x.\lambda k.\llbracket e \rrbracket^{\text{1CExp}}_{\text{name}} k && \text{– where } k \text{ is fresh} \\ \llbracket \lambda^1 x.\lambda k.e \rrbracket^{\text{1Answer}}_{\text{name}} &= \lambda x.\lambda k.\llbracket e \rrbracket^{\text{1CExp}}_{\text{name}} k \\ \llbracket \text{error} \rrbracket^{\text{1Answer}}_{\text{name}} &= \text{error} \end{aligned}$$

Fig. 12. Translation from 1CPS to CPS – naming continuation identifiers

$$\frac{\bullet \vdash^{\mathrm{1CExp}}_{\mathrm{1cc}} e \hookrightarrow a}{\vdash^{\mathrm{1CProg}}_{\mathrm{1cc}} \lambda\star.e \hookrightarrow a} \qquad \frac{\bullet \vdash^{\mathrm{1CExp}}_{\mathrm{1cc}} e[\mathrm{swap}\,\bullet/k] \hookrightarrow a}{\vdash^{\mathrm{1CProg}}_{\mathrm{1cc}} \lambda^{1} k.e \hookrightarrow a}$$

$$\frac{}{\varphi \vdash^{\mathrm{1CExp}}_{\mathrm{1cc}} \ell\, t\, c \hookrightarrow \mathrm{error}}$$

$$\frac{\varphi \vdash^{\mathrm{1CExp}}_{\mathrm{1cc}} e[t/x] \hookrightarrow a}{\varphi \vdash^{\mathrm{1CExp}}_{\mathrm{1cc}} (\lambda x.\lambda\star.e)\, t\, \star \hookrightarrow a} \qquad \frac{\varphi,\, \lambda v.e' \vdash^{\mathrm{1CExp}}_{\mathrm{1cc}} e[t/x] \hookrightarrow a}{\varphi \vdash^{\mathrm{1CExp}}_{\mathrm{1cc}} (\lambda x.\lambda\star.e)\, t\, \lambda v.e' \hookrightarrow a}$$

$$\frac{\varphi \vdash^{\mathrm{1CExp}}_{\mathrm{1cc}} e[t/x,\, \mathrm{swap}\, \varphi/k] \hookrightarrow a}{\varphi \vdash^{\mathrm{1CExp}}_{\mathrm{1cc}} (\lambda x.\lambda^{1} k.e)\, t\, \star \hookrightarrow a} \qquad \frac{\varphi,\, \lambda v.e' \vdash^{\mathrm{1CExp}}_{\mathrm{1cc}} e[t/x,\, \mathrm{swap}\, (\varphi,\, \lambda v.e')/k] \hookrightarrow a}{\varphi \vdash^{\mathrm{1CExp}}_{\mathrm{1cc}} (\lambda x.\lambda^{1} k.e)\, t\, \lambda v.e' \hookrightarrow a}$$

$$\frac{\varphi' \vdash^{\mathrm{1CExp}}_{\mathrm{1cc}} e[t/x] \hookrightarrow a}{\varphi \vdash^{\mathrm{1CExp}}_{\mathrm{1cc}} (\lambda x.\lambda\star.e)\, t\, (\mathrm{swap}\, \varphi') \hookrightarrow a} \qquad \frac{\varphi' \vdash^{\mathrm{1CExp}}_{\mathrm{1cc}} e[t/x,\, \mathrm{swap}\, \varphi'/k] \hookrightarrow a}{\varphi \vdash^{\mathrm{1CExp}}_{\mathrm{1cc}} (\lambda x.\lambda^{1} k.e)\, t\, (\mathrm{swap}\, \varphi') \hookrightarrow a}$$

$$\frac{\varphi \vdash^{\mathrm{1CExp}}_{\mathrm{1cc}} e[t/v] \hookrightarrow a}{\varphi \vdash^{\mathrm{1CExp}}_{\mathrm{1cc}} (\lambda v.e)\, t \hookrightarrow a} \qquad \frac{}{\bullet \vdash^{\mathrm{1CExp}}_{\mathrm{1cc}} \star t \hookrightarrow t} \qquad \frac{\varphi \vdash^{\mathrm{1CExp}}_{\mathrm{1cc}} e[t/v] \hookrightarrow a}{\varphi,\, \lambda v.e \vdash^{\mathrm{1CExp}}_{\mathrm{1cc}} \star t \hookrightarrow a}$$

$$\frac{}{\varphi \vdash^{\mathrm{1CExp}}_{\mathrm{1cc}} \mathrm{swap}\, \bullet\, t \hookrightarrow t} \qquad \frac{\varphi' \vdash^{\mathrm{1CExp}}_{\mathrm{1cc}} e[t/v] \hookrightarrow a}{\varphi \vdash^{\mathrm{1CExp}}_{\mathrm{1cc}} \mathrm{swap}\, (\varphi',\, \lambda v.e)\, t \hookrightarrow a}$$

Fig. 13. Stack machine for 1CPS programs

Figure 13 displays a stack-based abstract machine for 1CPS programs. This machine is a version of the stack machine of Section 3 where the substitution for continuation identifiers occurring in second-class position or not occurring at all is implemented with a global control stack (as in Figure 8), and where the substitution for continuation identifiers occurring in first-class position is implemented by copying the stack into a swap form (which is new).

Calls: When a function declaring a second-class continuation is applied, its continuation is pushed on φ. When a function declaring a first-class continuation is applied, its continuation is also pushed on φ and the resulting new stack is copied into a swap form.

Returns: When a continuation is needed, it is popped from φ. If φ is empty, the intermediate result sent to the continuation is the final answer. When a swap form is encountered, its copy of φ is restored.

More formally, the judgment

$$\vdash^{\mathrm{1CProg}}_{\mathrm{1cc}} p \hookrightarrow a$$

is satisfied whenever a CPS program $p \in$ 1CProg evaluates to an answer $a \in$ 1Answer. The auxiliary judgment

$$\varphi \vdash^{\mathrm{1CExp}}_{\mathrm{1cc}} e \hookrightarrow a$$

is satisfied whenever an expression $e \in$ 1CExp evaluates to an answer a, given a control stack $\varphi \in$ 1CStack. The machine starts and stops with an empty control stack.

We prove the equivalence between the stack machine and the standard machine as in Section 3.2.

Theorem 2 (Simulation). *The stack machine of Figure 13 and the standard machine are equivalent:*

1. $\vdash^{\text{CProg}}_{\mathit{std}} p \hookrightarrow a$ *if and only if* $\vdash^{\text{1CProg}}_{\mathit{1cc}} [\![p]\!]^{\text{CProg}}_{\text{strip}} \hookrightarrow [\![a]\!]^{\text{Answer}}_{\text{strip}}$.
2. $\vdash^{\text{CExp}}_{\mathit{std}} [\![e]\!]^{\text{CExp}}_{\text{strip}} k\{\varphi\}_1 \hookrightarrow a$ *if and only if* $\varphi \vdash^{\text{1CExp}}_{\mathit{1cc}} [\![e]\!]^{\text{CExp}}_{\text{strip}} k \hookrightarrow [\![a]\!]^{\text{Answer}}_{\text{strip}}$*, for some* k*.*

Proof. Similar to the proof of Theorem 1. □

4.3 Summary and Conclusion

We have formalized and proven correct a stack machine for CPS programs with first-class continuations. This machine is idealized in that, e.g., it has no provision for stack overflow. Nevertheless, it embodies the most classical implementation strategy for first-class continuations: the stack is copied at call/cc time, i.e., in the CPS world, when a first-class continuation identifier is declared; and conversely, the stack is restored at throw time, i.e., in the CPS world, when a first-class continuation identifier is invoked. This design keeps second-class continuations costless – in fact it is a zero-overhead strategy in the sense of Clinger, Hartheimer, and Ost [4, Section 3.1]: only programs using first-class continuations pay for them.

Furthermore, and as in Section 3, our representation of φ embodies its LIFO nature without committing to an actual representation. This representation can be retentive (in which case φ is implemented as a pointer into the heap) or destructive (in which case φ is implemented as, e.g., a rewriteable array) [3]. In both cases, swap φ is implemented as copying φ. Copying the pointer yields captured continuations to be shared and copying the array yields multiple representations of captured continuations.

5 A Segmented Stack Machine for First-Class Continuations

Coroutines and threads are easily simulated using call/cc, but these simulations are allergic to representing control as a rewriteable array. Indeed for every switch this array is copied in the heap, yielding multiple copies to coexist without sharing, even though these copies are mostly identical.

Against this backdrop, implementations such as PC Scheme [2] segment the stack, using the top segment as a stack cache: if this cache overflows, it is flushed to the heap and the computation starts afresh with an empty cache; and if it underflows, the last flushed cache is restored. Flushed caches are linked LIFO in the heap.[3] A segmented stack accomodates call/cc and throw very simply: at call/cc time, the cache is flushed to the heap and a pointer to it is retained; and

[3] If the size of the stack cache is one, the segmented implementation coincides with a heap implementation.

at throw time, the flushed cache that is pointed to is restored. As for the bulk of the continuations, it is not copied but shared between captured continuations.

It is simple to expand the stack machine of Section 4 into a segmented stack machine. One simply needs to define the judgment

$$\Phi\,;\varphi \vdash^{\mathrm{CExp}}_{\mathrm{1cc}'} e \hookrightarrow a$$

where φ, e, and a are in Section 4 and Φ denotes a LIFO list of φ's. (One also needs an overflow predicate for φ.)

Thus equipped, it is also simple to expand the stack substitution of Section 4, and to state and prove a simulation theorem similar to Theorem 2, thereby formalizing what Clinger, Hartheimer, and Ost name the "chunked-stack strategy" [4]. Another moderate effort makes it possible to formalize the author's incremental garbage collection of unshared continuations by one-bit reference counting [5]. One is also in position to formalize "one-shot continuations" [14].

Acknowledgments: I am grateful to Belmina Dzafic and Frank Pfenning for our joint work, which forms the foundation of the present foray. Throughout, and as always, Andrzej Filinski has been a precious source of sensible comments and suggestions. This article has also benefited from the interest and comments of Lars R. Clausen, Daniel Damian, Bernd Grobauer, Niels O. Jensen, Julia L. Lawall, Lasse R. Nielsen, Morten Rhiger, and Zhe Yang. I am also grateful for the opportunity to have presented this work at Marktoberdorf, at the University of Tokyo, and at KAIST in the summer and in the fall of 1999. Finally, thanks are due to the anonymous referees for stressing the issue of retention vs. deletion.

References

1. Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In *Third International Symposium on Programming Language Implementation and Logic Programming*, number 528 in Lecture Notes in Computer Science, pages 1–13, Passau, Germany, August 1991.
2. David B. Bartley and John C. Jensen. The implementation of PC Scheme. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 86–93, Cambridge, Massachusetts, August 1986.
3. Daniel M. Berry. Block structure: Retention or deletion? (extended abstract). In *Conference Record of the Third Annual ACM Symposium on Theory of Computing*, pages 86–100, Shaker Heights, Ohio, May 1971.
4. William Clinger, Anne H. Hartheimer, and Eric M. Ost. Implementation strategies for first-class continuations. *Higher-Order and Symbolic Computation*, 12(1):7–45, 1999.
5. Olivier Danvy. Memory allocation and higher-order functions. In *Proceedings of the ACM SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques*, SIGPLAN Notices, Vol. 22, No 7, pages 241–252, Saint-Paul, Minnesota, June 1987.
6. Olivier Danvy, Belmina Dzafic, and Frank Pfenning. On proving syntactic properties of CPS programs. In *Third International Workshop on Higher-Order Operational Techniques in Semantics*, volume 26 of *Electronic Notes in Theoretical Computer Science*, pages 19–31, Paris, France, September 1999.

7. Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.
8. Olivier Danvy and Julia L. Lawall. Back to direct style II: First-class continuations. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. V, No. 1, pages 299–310, San Francisco, California, June 1992.
9. Olivier Danvy and Frank Pfenning. The occurrence of continuation parameters in CPS terms. Technical report CMU-CS-95-121, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, February 1995.
10. Bruce F. Duba, Robert Harper, and David B. MacQueen. Typing first-class continuations in ML. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 163–173, Orlando, Florida, January 1991.
11. Belmina Dzafic. Formalizing program transformations. Master's thesis, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 1998.
12. Matthias Felleisen. *The Calculi of λ-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages.* PhD thesis, Department of Computer Science, Indiana University, Bloomington, Indiana, August 1987.
13. Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 28, No 6, pages 237–247, Albuquerque, New Mexico, June 1993.
14. Christopher T. Haynes and Daniel P. Friedman. Embedding continuations in procedural objects. *ACM Transactions on Programming Languages and Systems*, 9(4):582–598, 1987.
15. Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 25, No 6, pages 66–77, White Plains, New York, June 1990.
16. Richard A. Kelsey and Jonathan A. Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4):315–336, 1994.
17. David Kranz, Richard Kesley, Jonathan Rees, Paul Hudak, Jonathan Philbin, and Norman Adams. Orbit: An optimizing compiler for Scheme. In *Proceedings of the 1986 Symposium on Compiler Construction*, SIGPLAN Notices, Vol. 21, No 7, pages 219–233, Palo Alto, California, June 1986.
18. Peter J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.
19. Drew McDermott. An efficient environment allocation scheme in an interpreter for a lexically-scoped Lisp. In *Conference Record of the 1980 LISP Conference*, pages 154–162, Stanford, California, August 1980.
20. Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.
21. Guy L. Steele Jr. Rabbit: A compiler for Scheme. Technical Report AI-TR-474, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978.
22. Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. *Higher-Order and Symbolic Computation*, 13(1/2), 2000. Reprint of the technical monograph PRG-11, Oxford University Computing Laboratory (1974).

Correctness of Java Card Method Lookup via Logical Relations

Ewen Denney and Thomas Jensen

Projet Lande, IRISA, Rennes Cedex 35042, France

Abstract. We formalise the Java Card bytecode optimisation from class file to CAP file format as a set of constraints between the two formats, and define and prove its correctness. Java Card bytecode is formalised as an abstract operational semantics, which can then be instantiated into the two formats. The optimisation is given as a logical relation such that the instantiated semantics are observably equal. The proof has been automated using the Coq theorem prover.

Using a high-level language for programming embedded systems may require a transformation phase in order that the compiled code fits on the device. In this paper we describe a method for formally proving the correctness of such a transformation. The method makes extensive use of types to describe the various run-time structures and relies on the notion of logical relation to relate the two representations of the code. We present the method in the setting of mapping Java onto smart cards. The Java Card language [10] is a trimmed down dialect of Java aimed at programming smart cards. As with Java, Java Card is compiled into bytecode, which is then verified and executed on a virtual machine [4], installed on a chip on the card itself. However, the memory and processor limitations of smart cards necessitate a further stage, in which the bytecode is optimised from the standard class file format of Java, to the *CAP file* format [11]. The core of this optimisation is a *tokenisation* in which names are replaced with tokens enabling a faster lookup of various entities.

We describe a semantic framework for proving the correctness of Java Card tokenisation. The basic idea is to give an abstract description of the constraints given in the official specification of the tokenisation and show that any transformation satisfying these constraints is 'correct'. This is independent of showing that there actually exists a collection of functions satisfying these constraints. This article concentrates on proving the correctness of the specification. The formal development of an algorithm is the subject of another report. The main advantage of decoupling 'correctness' into two steps is that we get a more general result: rather than proving the correctness of one particular algorithm, we are able to show that the constraints described in Sun's official specification [11] (given certain assumptions) are sufficient. We give a formalisation and correctness proof for the part concerned with dynamic method lookup. A comprehensive formalisation appears as a technical report [2].

1 The Conversion

Java source code is compiled on a class by class basis into the *class file* format. By contrast, Java Card *CAP files* correspond to packages. They are produced by the *con-*

G. Smolka (Ed.): ESOP/ETAPS 2000, LNCS 1782, pp. 104–118, 2000.

version of a collection of class files. In the class file format, methods, fields and so on are referred to using strings. In CAP files, however, tokens are ascribed to the various entities. The idea is that if a method, say, is publically visible[1], then it is ascribed a token. If the method is only visible within its package, then it is referred to directly using an offset into the relevant data structure. Thus references are either internal or external. The conversion groups entities from different class files into the components of a CAP file. For example, all constant pools of the class files forming a package are merged into one constant pool component, and all method implementations are gathered in the same method component. One significant difference between the two formats is the way in which the method tables are arranged. In a class file, the methods item contains all the information relevant to methods defined in that class. In the CAP file, this information is shared between the class and method components. The method component contains the implementation details (*i.e.* the bytecode) for the methods defined in this package. The class component is a collection of class information structures. Each of these contains separate tables for the package and public methods, mapping method tokens to offsets into the method component. The method tables contain the information necessary for resolving any method call in that class.

The conversion is presented in [11] as a collection of constraints on the CAP file, rather than as an explicit mapping between class and CAP formats. For example, if a class inherits a method from a superclass then the conversion can choose to include the method token in the relevant method table or, instead, that the table of the superclass should be searched. There is a choice, therefore, between copying all inherited methods, or having a more compressed table. The specification does not constrain this choice. We adopt a simplified definition of the conversion, only considering classes, constant pools, and methods (with inheritance and overwriting). In particular, we ignore fields, exceptions and interfaces. The conversion also includes a number of mandatory optimisations such as the inlining of final fields, and the type-based specialisation of instructions [10,11], which we do not treat here.

2 Overview of Formalisation

The conversion from class file to CAP format is a transformation between formats of two virtual machines. The first issue to be addressed is determining in what sense, exactly, the conversion to token format should be regarded as an equivalence. We cannot simply say that the JVM and JCVM have the same behaviour for all bytecodes, in class and CAP file format respectively, because, *a priori*, the states of the virtual machines are themselves in different formats. Instead, we adopt a simple form of equivalence based on the notion of *representation independence* [5]. This is expressed in terms of so-called *observable* types. This limits us to comparing the two interpretations in terms of words, but this is sufficient to observe the operand stack and local variables, where the results of execution are stored.

Representation independence may be proven by defining *coupling relations* between the two formats that respect the tokenisation and are the identity at observable types.

[1] We follow the terminology of [11], where a method is *public visible* if it has either a `protected` or a `public` modifier, and *package visible* if it is declared `private` or has no visibility modifier.

This can be seen as formalising a *data refinement* from class to CAP file. We formalise the relations nondeterministically as any family of relations that satisfies certain constraints, rather than as explicit transformations. This is because there are many possible tokenisations and we wish to prove any reasonable optimisation correct.

The virtual machines are formalised in an operational style, as transition relations over abstract machines. We adopt the action semantics formalism of Mosses [6], using a mixture of operational and denotational styles: the virtual machines are formalised operationally, parameterised with respect to a number of auxiliary functions, which are then interpreted denotationally. This modular presentation of the semantics facilitates the comparison between the two formats. We illustrate this for dynamic method lookup, used in the semantics of the method invocation instructions. The lookup function which searches for the implementation of a method is dependent on the layout of the method tables. The operational rule giving the semantics of the method invocation instructions, presented in Section 5, is parameterised with respect to the lookup function. Then in Section 6 two possible interpretations of lookup are given.

In Section 4, we define abstract types for the various entities converted during tokenisation, which are common to the two formats. For example, $\texttt{Class\_ref}$ and $\texttt{Environment}$. It is this type structure which is used to define the logical relations. In Section 5 we give an operational semantics which is independent of the underlying class/CAP file format. The structure of the class/CAP file need not be visible to the operational semantics. We need only be able to extract certain data corresponding to a particular method, such as the appropriate constant pool. In Section 6, we give the specific details of the class file and CAP file formats. defined as interpretations of types and auxiliary functions, $[\![\cdot]\!]_{name}$ and $[\![\cdot]\!]_{tok}$. We refer to these as the *name* and the *token* interpretation, respectively.

In Section 7, we define the logical relation, $\{R_\theta\}_{\theta \in Abstract\_type}$. It is convenient to group the definition into several levels. First, there are various basic observable types (byte, short, *etc.*), γ, for which we have $R_\gamma = id_\gamma$. Second, there are the references, ι, such as package and class references, for which the relation R_ι represents the tokenisation of named items. Third, the constraints on the organisation into components (which we will call the *componentisation*) are expressed in R_κ, where κ includes method information structures, constant pools, and so on. This represents the relationship between components in CAP files and the corresponding entities in class files. Using the above three families of relations we can define R_θ for each type, θ, where

$$\theta ::= \gamma \mid \iota \mid \kappa \mid \theta \times \theta' \mid \theta \to \theta' \mid \theta + \theta' \mid \theta^*.$$

The family of relations, $\{R_\theta\}_{\theta \, \in \, Abstract\_type}$, represents the overall construction of components in the CAP file format from a class file. The relations are ‘logical’ in the sense that the definitions for defined types follow automatically. For example, we define the type of the environment that contains the class hierarchy as

$$\texttt{Environment} = \texttt{Package\_ref} \to \texttt{Package}.$$

and so the definition of $R_{\texttt{Environment}}$ follows from those of $R_{\texttt{Package\_ref}}$, $R_{\texttt{Package}}$ and the standard construction of $R_{\_ \to \_}$; similarly for $R_{\texttt{Heap}}$.

3 Related Work

There have been a number of formalisations of the Java Virtual Machine which have some relevance for our work here on Java Card. Bertelsen [1] gives an operational semantics which we have used as a starting point. He also considers the verification conditions, which considerably complicates the rules, however. Pusch has formalised the JVM in HOL [8]. Like us, she considers the class file to be well-formed so that the hypotheses of rules are just assignments. The operational semantics is presented directly as a formalisation in HOL, whereas we have chosen (equivalently) to use inference rules. All these works make various simplifications and abstractions. However, since these are formalisations of Java rather than Java Card they do not consider the CAP file format. In contrast, the work of Lanet and Requet [3] is specifically concerned with Java Card. They also aim to prove the correctness of Java Card tokenisation. Their work can be seen as complementing ours. They concentrate on optimisations, including the type specialisation of instructions, and do not consider the conversion as such. In contrast, we have specified the conversion but ignored the subsequent optimisations. Their formalism is based on the B method, so the specification and proof are presented as a series of refinements. In [7], Pusch proves the correctness of an implementation of Prolog on an abstract machine, the WAM. The proof structure is similar to ours, although there are refinements through several levels. There are operational semantics for each level, and correctness in expressed in terms of equivalence between levels. The differences between the semantics are significant, since they are not factored out into auxiliary functions as here. She uses a big-step operational semantics, which is not appropriate for us because we wish to compare intermediate results. Moreover, she uses an abstraction function on the initial state, the results being required to be identical, whereas we have a *relation* for both initial and final states.

4 Abstract Types

We use types to structure the transformation. These are not the types of the Java Card language, but rather are based on the simply-typed lambda calculus with sums, products and lists. We use record types with the actual types of fields (drawn from the official specification where not too confusing) serving as labels. Occasionally we use terms as singleton types, such as `0xFFFF` and `0`. There are two sorts of types: *abstract* and *concrete*. The idea is that abstract types are those we can think of independently of a particular format. The concrete types are the particular realisations of these, as well as types which only make sense in one particular model. For example, `CP_index` is the abstract type of indices into a constant pool for a given package. In the name interpretation, this is modelled by a class name and an index into the constant pool of the corresponding class file, *i.e.* $\texttt{Class\_name} \times \texttt{Index}$ where `Index` is a concrete type. In the token interpretation, however, since all the constant pools are merged, we have $[\![\texttt{CP\_index}]\!]_{tok} = \texttt{Package\_tok} \times \texttt{Index}$. Another example is the various distinctions that are made between method and field references in CAP files, but not class files, and which are not relevant at the level of the operational semantics, which concerns terms of abstract types. We arrange the types so that as much as possible is common between the two formats. For example, it is more convenient to uniformly define environments as mappings of type $\texttt{Package\_ref} \to \texttt{Package}$, with Package interpreted as $\texttt{Class\_name} \to \texttt{Class\_file}$ or CAP_file.

There is a 'type of types' for the two forms of data type in Java Card — primitive types, *i.e.*, the simple types supported directly on the card, and reference types.

$$\texttt{Type} = \{\texttt{Boolean}, \texttt{Byte}, \texttt{Short}\} + \texttt{Reference\_type}.$$
$$\texttt{Reference\_type} = \texttt{Array\_type} + \texttt{Class\_ref}.$$

We use a separate type, Object_ref, to refer to objects on the heap. The objects themselves contain a reference to the appropriate class or array of which they form an instance.

The type Word is an abstract unit of storage and is platform specific. All we need know is that object references and the basic types, Byte, Short and Boolean, can be stored in a Word. Rather than use an explicit coercion, we assume

$$\texttt{Word} = \texttt{Object\_ref} + \texttt{Null} + \texttt{Boolean} + \texttt{Byte} + \texttt{Short}.$$

Thus a word is (*i.e.* represents) either a reference (possibly null) or an element of a primitive type. Furthermore, we define $\texttt{Value} = \texttt{Word}$. Although this is not strictly necessary, there is a conceptual distinction. If we were to introduce values of type int, then a value could be either a word or a double word.

There are several forms of references used during tokenisation, *viz.*, Package_ref, Class_ref and Method_ref. We distinguish Package from Package_ref, and similarly for the other items. Note that a *reference* is a composite entity which can be context dependent (*e.g.* in the CAP format a class reference can be in internal or external forms). We assume, however, that sufficient information is given so that references make sense globally. For example, class names are fully qualified, and class tokens are paired with a package token. We take field and method references to be to particular members of some class, and so contain a class reference. In contrast, an *identifier* is a name or a token (these are not used at the abstract level though). Using these basic types, we can then construct complex types using the usual type constructors: (non-dependent) sum, product, function and list types (denoted θ^*) as we did when defining the environment at the end of Sect. 2.

5 Operational Semantics

We define an operational semantics framework that allows us to model the execution of both class and CAP files. This is obtained by parameterising the semantics on a number of *auxiliary functions* that embody the differences between the two formats. This factorisation of the semantics reduces the equivalence proof considerably.

The official specification of the JCVM (and JVM) is given in terms of *frames*. A frame represents the state of the current method invocation, together with any other useful data. We introduce the notion of *configuration*, consisting of (the abstract syntax of) the code of the current method still to be executed, the operand stack, the local variables, and the current class reference. We write these as $\texttt{Config}\,(b, o, l, c)$ or just $\langle b, o, l, c\rangle$. To account for method invocations, we allow a configuration itself to be considered as an instruction. When a method is invoked, the current instruction becomes a new configuration. Instead of a stack of frames, then, we have a single piece of 'code' (in this general sense). This form of closure is more general than the traditional idea of a call stack but helps simplify the proof. Method invocation is modelled by replacing the invoking instruction with a configuration that contains the code of the invoked method (see the detailed description of `invokevirtual` below). Execution of a method body is modelled by allowing transitions inside a configuration.

$$\frac{f \Rightarrow f'}{\langle \texttt{Config}\ f, ops, l, c\rangle \Rightarrow \langle \texttt{Config}\ f', ops, l, c\rangle}$$

The method invocation instructions (and others) take an argument which is an index into either the constant pool of a class file, or into the constant pool component of a CAP file. This means that the 'concrete' bytecode is itself dependent on the implementation and is therefore modelled by an abstract type. Formally, we define a transition relation

$$\Rightarrow\ \subseteq\ \texttt{Config} \times \texttt{Arrow} \times \texttt{Config}$$

using the types

$$\begin{aligned}
\texttt{Config} &= \texttt{Bytecode} \times \texttt{Word}^* \times \texttt{Locals} \times \texttt{Class\_ref}\\
\texttt{Arrow} &= \texttt{Global\_state} \rightarrow \texttt{Global\_state}\\
\texttt{Global\_state} &= \texttt{Environment} \times \texttt{Heap}\\
\texttt{Bytecode} &= \texttt{Instruction} + (\texttt{Bytecode} \times \texttt{Bytecode}) + \texttt{Config}
\end{aligned}$$

As mentioned above, the structure of the class/CAP file need not be visible to the operational semantics. We use a number of auxiliary functions, some of which have preconditions that we take as concomitant with the well-formedness of the class file. The definition of method invocation uses the `lookup` function

$$\texttt{lookup} : \texttt{Class\_ref} \times \texttt{Method\_ref} \rightarrow \texttt{Class\_ref} \times \texttt{Bytecode}$$

that takes the actual class reference, together with the declared method reference (which contains the class where the method is declared), and returns the class reference where the method is defined together with the code. Function $\texttt{method\_nargs} : \texttt{Method\_ref} \rightarrow \texttt{Nat}$ returns the number of arguments for a given method reference. The instruction for virtual method invocation is evaluated as follows:

1. The two byte index, i, into the constant pool is resolved to get the declared method reference containing the declared class reference and a method identifier (either a signature or token).
2. The number of arguments to the method is calculated.
3. The object reference, r, is popped off the operand stack.
4. Using the heap, we get $\mathit{heap}(r) = \langle \mathit{act\_cref}, \_\rangle$, the actual class reference (fully qualified name or a package/class token pair).
5. We then do $\texttt{lookup}(\mathit{act\_cref}, \mathit{dec\_mref})$, getting the class where the method is implemented, and its bytecode. The lookup function is used with respect to the class hierarchy (environment).
6. A configuration is created for this method and evaluation proceeds from there.

$$\frac{\begin{array}{lr} \mathit{dec\_mref} := \texttt{constant\_pool}\ (c)(i) & \text{get declared method reference}\\ n := \texttt{method\_nargs}(\mathit{stat\_mref}) & \text{get number of arguments}\\ \langle \mathit{act\_cref}, \_\rangle := \mathit{heap}(r) & \text{get actual class reference from heap}\\ \langle m\_cl, m\_cd\rangle := \texttt{lookup}(\mathit{act\_cref}, \mathit{dec\_mref}) & \text{look up method} \end{array}}{\langle \texttt{invokevirtual}\ i, a_1 \dots a_n :: r :: s, l, c\rangle \Rightarrow \langle\langle m\_cd, \langle\rangle, a_1 \dots a_n :: r, m\_cl\rangle, s, l, c\rangle}$$

In the following sections we show how to instantiate the semantic framework (in particular the `lookup` function) to obtain a class file and a CAP file semantics.

6 Interpretations

The name interpretation gives semantics using Java class files (see Figure 1). Since this is fairly standard we give a brief description. Classes are described by fully qualified names, whereas methods and fields are given signatures, consisting of an unqualified name and a type, together with the class of definition. We assume a function $\mathit{pack\_name}$ which gives the package name of a class name. The data is arranged into class files, each of which contains all the information corresponding to a particular class. We only give the interpretation of those parts used here. We group the class files by package into a global environment so $\mathit{env\_name}(p)(c)$ denotes the class file in package p with name c.

$$\llbracket \texttt{Package} \rrbracket_{name} = \texttt{Class\_name} \rightarrow \texttt{Class\_file}$$

$$\llbracket \texttt{Class\_ref} \rrbracket_{name} = \texttt{Class\_name}$$

$$\llbracket \texttt{Method\_ref} \rrbracket_{name} = \texttt{Class\_name} \times \texttt{Sig}$$

$$\texttt{Sig} = \texttt{Method\_name} \times \llbracket \texttt{Type} \rrbracket^{*}_{name}$$

$$\llbracket \texttt{Class} \rrbracket_{name} = \texttt{Class\_file}$$

$$\texttt{Class\_file} = \texttt{Class\_flags} \times \texttt{Super} \times \texttt{Methods\_item} \times \texttt{Constant\_pool\_item} \times \texttt{Class\_name}$$

$$\texttt{Super} = \texttt{Class\_name} + \texttt{Void}$$

$$\llbracket \texttt{Pack\_methods} \rrbracket_{name} = \texttt{Class\_name} \rightarrow \texttt{Methods\_item}$$

$$\texttt{Methods\_item} = \texttt{Sig} \rightarrow \texttt{Method\_info}$$

$$\begin{array}{l} \texttt{Method\_info} = \\ \texttt{Method\_flags} \times \texttt{Sig} \times (\llbracket \texttt{Type} \rrbracket_{name} + \texttt{Void}) \times \texttt{Maxstack} \times \texttt{Maxlocals} \times \texttt{Bytecode} \end{array}$$

Fig. 1. Name Interpretation

Method signatures are not considered to include the return type. We assume that the signature in the result of a methods item is the same as the argument.

There are a number of possibilities for how method lookup should be defined, depending on the definition of inheritance. For example, [1,8] use a ‘naive’ lookup which does not take account of visibility modifiers. A fuller discussion of this appears in [9].

In the JCVM, data is arranged by packages into CAP files. Each CAP file consists of a number of components, but not all are used for method lookup (or, indeed, the rest

```
lookup_name (act_class, (sig, dec_class)) =
let dec_pk  = pack_name(dec_class)
     act_pk  = pack_name(act_class)
     (_,_,meth_dec,_,_) = env_name (dec_pk) (dec_class)
     (_,super,_,meth_act,_,_)  = env_name (act_pk) (act_class)
     (dec_flags,_,_,_,_,_) = meth_dec(sig)  in
if meth_act(sig) = undefined
then lookup_name(super, (sig, dec_class))
else if
   dec_flags(protected) or dec_flags(public) or act_pk = act_pk
   then let (_,_,_,_,_,code) = meth_act(sig) in (act_class,code)
   else lookup_name(super, (sig, dec_class))
```

Fig. 2. The `lookup` function for the class file format.

of the operational semantics). We just include those components we need here, namely, the constant pool, class and method components.

References to items external to a package are via tokens — for packages, classes, and virtual methods — each with a particular range and scope. These are then used to find internal offsets into the components. For example, a class reference is either an internal offset into the class component of the CAP file of the class' package, or an external reference composed of a package token and a class token. However, since we need to relate the reference to class names, we will assume that all references come with package information, even though this is superfluous in the case of internal references.

$$
\begin{gathered}
[\![\mathtt{Package}]\!]_{tok} = \mathtt{CAP\_file} \\
\mathtt{CAP\_file} = \mathtt{Constant\_pool\_comp} \times \mathtt{Class\_comp} \times \mathtt{Method\_comp} \\
[\![\mathtt{Package\_ref}]\!]_{tok} = \mathtt{Package\_tok} \\
[\![\mathtt{Class\_ref}]\!]_{tok} = \mathtt{Package\_tok} \times (\mathtt{Class\_tok} + \mathtt{Offset}) \\
[\![\mathtt{Method\_ref}]\!]_{tok} = [\![\mathtt{Class\_ref}]\!]_{\mathrm{tok}} \times \mathtt{Virtual\_method\_tok} \\
[\![\mathtt{Class}]\!]_{tok} = \mathtt{Class\_info} \\
\mathtt{Class\_comp} = \mathtt{Offset} \rightarrow \mathtt{Class\_info} \\
\mathtt{Class\_info} = \mathtt{Class\_flags} \times \mathtt{Super} \times \mathtt{Public\_table} \times \mathtt{Package\_table} \times \mathtt{Class\_ref} \\
\mathtt{Public\_table} = \mathtt{Public\_base} \times \mathtt{Public\_size} \times (\mathtt{Index} \rightarrow \mathtt{Offset} + \{\mathtt{0xFFFF}\}) \\
\mathtt{Package\_table} = \mathtt{Package\_base} \times \mathtt{Package\_size} \times (\mathtt{Index} \rightarrow \mathtt{Offset}) \\
[\![\mathtt{Pack\_methods}]\!]_{tok} = \mathtt{Method\_comp} \\
\mathtt{Method\_comp} = \mathtt{Offset} \rightarrow \mathtt{Method\_info} \\
\mathtt{Method\_info} = \mathtt{Method\_flags} \times \mathtt{Maxstack} \times \mathtt{Nargs} \times \mathtt{Max\_locals} \times \mathtt{Bytecode}
\end{gathered}
$$

Fig. 3. Token Interpretation

The class component consists of a list of class information structures, each of which has method tables, giving offsets into the method component, where the method implementations are found. The lookup algorithm uses tokens to calculate the corresponding method table index. There are separate tables for public and package methods. Method access information is given implicitly by the tokens rather than by flags. The two method tables each contain a base, size and 'list' of entries. The entries are defined from the $base$ to $base + size - 1$ inclusive. The entry for a public method will be `0xFFFF` if the method is defined in another package.

For a given class reference, the function `class_info` finds the corresponding class information structure in the global environment. The variant, `class_info`$'$ returns the class information structure in a particular CAP file. The function `method_array` simply finds the method component for a given class reference. We assume the existence of functions `class_offset` and `method_offset` for resolving external tokens to internal offsets It follows from the definition of the abstract type `Environment`, that the environment in the token format consists of a mapping from package tokens to their corresponding CAP file *i.e.*, $env_{tok} : \texttt{Package\_tok} \rightarrow \texttt{CAP\_file}$. The lookup function takes a class reference (the declared class), a method reference (in the actual class), and returns the reference to the class where the code is defined, together with the bytecode itself. The main steps of the algorithm (see Fig. 4) are:

1. Get method array for the package of the actual class.
2. Get class information for the actual class.
3. If public: if defined then *get info* else *lookup super*.
   If package: if defined and visible then *get info* else *lookup super*.

7 Formalisation of Equivalence

We formalise the equivalence between the class and CAP formats as a family of relations, $\{R_\theta : [\![\theta]\!]_{name} \leftrightarrow [\![\theta]\!]_{tok}\}_{\theta \in Abstract\_type}$ indexed by abstract type, θ. The idea is that $x\ R_\theta\ y$ when y is a *possible* transformation of x. The relations are not necessarily total, *i.e.* for some $x : [\![\theta]\!]_{name}$, there may not be a y such that $x\ R_\theta\ y$. Formally, the relations are defined as a mutually inductive collection of constraints, R_θ, for each type θ, where the types, θ, are given by the grammar:

$$\begin{aligned}
\gamma &::= \texttt{Bool} \mid \texttt{Nat} \mid \texttt{Object\_ref} \mid \texttt{Boolean} \mid \texttt{Byte} \mid \texttt{Short} \mid \texttt{Value} \mid \texttt{Word} \\
\iota &::= \texttt{Package\_ref} \mid \texttt{Ext\_class\_ref} \mid \texttt{Class\_ref} \mid \texttt{Method\_ref} \\
\kappa &::= \texttt{CP\_index} \mid \texttt{CP\_info} \mid \texttt{Method\_info} \mid \texttt{Package} \mid \texttt{Class} \mid \\
&\quad\;\; \texttt{Constant\_pool} \mid \texttt{Pack\_methods} \\
\theta &::= \gamma \mid \iota \mid \kappa \mid \theta \times \theta' \mid \theta \rightarrow \theta' \mid \theta + \theta' \mid \theta^*
\end{aligned}$$

where the observable types are built up inductively from the γ, *i.e.* do not contain the ι and κ. There are two sources of underspecification. First, the relations really can be non-functional. Second, there is a choice for what some of the relations are. For example, $R_{\texttt{Class\_ref}}$ is *some* bijection satisfying certain constraints. The relations between the 'large' structures, however, are completely defined in terms of those between smaller ones. There are two parts to the transformation itself: the tokenisation, defined as the relations R_ι, and the 'componentisation', defined as the R_κ.

```
lookup_tok (act_class_ref, (dec_class_ref, method_tok)) =

let methods = method_array (act_class_ref)
    (_, super,(public_base,_, public_table),
              (package_base,_, package_table),_) : Class_info =
    class_info(act_class_ref) in
if method_tok div 128 = 0 then /* public */
   if method_tok >= public_base then
    let method_offset = public_table[method_tok-public_base] in
        if method_offset <>  0xFFFF
        then (act_class_ref, methods[method_offset].Bytecode)
        else /* look in superclass */
           lookup_tok(super, (dec_class_ref, method_tok))
   else /* look in superclass */
      lookup_tok(super, (dec_class_ref, method_tok))
else /* package */
   if method_tok >= package_base /\
      same_package(dec_class_ref, act_class_ref)
    then let method_offset =
         package_table[method_tok mod 128 - package_base]
      in (act_class_ref, methods[method_offset].Bytecode)
   else /* look in superclass */
      lookup_tok(super, (dec_class_ref, method_tok))
```

Fig. 4. The `lookup` function for the CAP file format.

7.1 Tokenisation

The relations, R_ι, represent the tokenisation of items. The general idea is to set up relations between the names and tokens assigned to the various entities, subject to certain constraints described in the specification.

In order to account for token scope, we relate names to tokens paired with the appropriate context information. For example, method tokens are scoped within a class, so the relation $R_{\texttt{Method\_ref}}$ is between pairs of class names and signatures, and pairs of class references and method tokens. We must add a condition, therefore, to ensure that the package token corresponds to the package name of this class name.

We assume that each of these relations is a bijection, modulo the equivalence between internal and external references (with one exception to account for the copying of virtual methods, explained below). Formally,

$$a\,R\,b\ \wedge\ a'\,R\,b\ \Rightarrow\ a = a'$$

$$a\,R\,b\ \Rightarrow\ (a\,R\,b'\ \iff\ \texttt{Equiv}(b, b'))$$

where equivalence, `Equiv`, of class references is defined as the reflexive symmetric closure of:

$$\texttt{Equiv}(\langle p\_tok, \mathit{offset}\rangle, \langle p\_tok, c\_tok\rangle)\ \iff\ \texttt{class\_offset}(p\_tok, c\_tok) = \mathit{offset}$$

The second condition contains two parts: that the relation is functional modulo `Equiv`, and that it is closed under `Equiv`. We say that R is an *external bijection* when these conditions hold. We extend the definition of `Equiv` and external bijection to the other references.

These relations are defined with respect to the environment (in name format). We use a number of abbreviations for extracting information from the environment. We write $c < c'$ for the subclass relation (*i.e.* the transitive closure of the direct subclass relation) and $\leq$ for its reflexive closure. In the token interpretation this is modulo `Equiv`. We write $m\_tok \in c\_ref$ when a method with token m_tok is declared in the class with reference c_ref, and $pack\_name(c)$ for the package name of the class named c.

We define function `Class_flag` for checking the presence of attributes such as `public`, `final`, *etc*. The tokenisation uses the notion of *external visibility*.

$$\texttt{Externally\_visible}(c\_name) = \texttt{Class\_flag}(c\_name, Public)$$

We will also write $public(sig)$ and $package(sig)$ according to the visibility of a method.

`Package_ref` : As mentioned above, we take package tokens to be externally visible. The relation $R_{\texttt{Package\_ref}}$ is simply defined as any bijection between package names and tokens.

`Ext_class_ref` : In order to define the relation for class references we first define the relation for external class references. We define $R_{\texttt{Ext\_class\_ref}}$ as a bijection between class names and external class references such that:
$c\_name\ R_{\texttt{Ext\_class\_ref}}\ (p\_tok, c\_tok) \Rightarrow$
$\texttt{Externally\_visible}(c\_name) \wedge\ pack\_name(c\_name)\ R_{\texttt{Package\_ref}}\ p\_tok$

`Method_ref` : This is not a bijection because of the possibility of copying. Although 'from names to tokens' we do have:

$$\begin{array}{l} \langle c\_name, sig\rangle\ R_{\texttt{Method\_ref}}\ \langle c\_ref, m\_tok\rangle \wedge \\ \langle c'\_name, sig'\rangle\ R_{\texttt{Method\_ref}}\ \langle c\_ref, m\_tok\rangle \end{array} \Rightarrow \begin{cases} c\_name = c'\_name \wedge \\ sig = sig' \end{cases}$$

for a converse we have:

$$\begin{array}{l} \langle c\_name, sig\rangle\ R_{\texttt{Method\_ref}}\ \langle c\_ref, m\_tok\rangle \wedge \\ \langle c\_name, sig\rangle\ R_{\texttt{Method\_ref}}\ \langle c'\_ref, m'\_tok\rangle \end{array} \Rightarrow \begin{cases} (c\_ref \leq c'\_ref \vee c'\_ref \leq c\_ref) \\ \wedge\ m\_tok = m'\_tok \end{cases}$$

The first condition says that if a method overrides a method implemented in a superclass, then it gets the same token. Restrictions on the language mean that overriding cannot change the method modifier from public to package or vice versa.

$$\left.\begin{array}{l} \langle c\_name, sig\rangle\ R_{\texttt{Method\_ref}}\ \langle c\_ref, m\_tok\rangle \wedge \\ \langle c'\_name, sig\rangle\ R_{\texttt{Method\_ref}}\ \langle c'\_ref, m'\_tok\rangle \wedge \\ c'\_name < c\_name \wedge \\ (package(sig) \Rightarrow same\_package(c\_name, c'\_name)) \end{array}\right\} \Rightarrow m\_tok = m'\_tok$$

The second condition says that the tokens for public introduced methods must have higher token numbers that those in the superclass. We assume a predicate, *new_method*, which holds of a method signature and class name when the method is defined in the class, but not in any superclass.

$$\begin{array}{l} public(sig)\ \wedge\ new\_method(sig, c\_name) \wedge \\ (c\_name, sig)\ R_{\texttt{Method\_ref}}\ (c\_ref, m\_tok) \end{array} \Rightarrow \begin{cases} \forall m'\_tok \in super(c\_ref)\,. \\ \qquad\qquad m\_tok > m'\_tok \end{cases}$$

Package-visible tokens for introduced methods are similarly numbered, if the superclass is in the same package.

$$\begin{array}{l} \mathit{package}(\mathit{sig}) \ \wedge \ \mathit{new\_method}(\mathit{sig}, \mathit{c\_name}) \ \wedge \\ (\mathit{c\_name}, \mathit{sig}) \ R_{\texttt{Method\_ref}} \ (\mathit{c\_ref}, \mathit{m\_tok}) \ \wedge \\ \mathit{same\_package}(\mathit{c\_name}, \mathit{super}(\mathit{c\_name})) \end{array} \Rightarrow \begin{cases} \forall m'\_tok \in \mathit{super}(\mathit{c\_ref})\,. \\ \qquad\qquad m\_tok > m'\_tok \end{cases}$$

The third condition says that public tokens are in the range 0 to 127, and package tokens in the range 128 to 255.

$$\begin{array}{l} \langle \mathit{c\_name}, \mathit{sig} \rangle \ R_{\texttt{Method\_ref}} \ \langle \mathit{c\_ref}, \mathit{m\_tok} \rangle \Rightarrow \\ \quad (\mathit{public}(\mathit{sig}) \Rightarrow 0 \le \mathit{m\_tok} \le 127) \ \wedge \ (\mathit{package}(\mathit{sig}) \Rightarrow 128 \le \mathit{m\_tok} \le 255) \end{array}$$

The specification [11] also says that tokens must be contiguously numbered starting at 0 but we will not enforce this.

7.2 Componentisation

The relations in the previous section formalise the correspondence between named and tokenised entities. When creating the CAP file components, all the entities are converted, including the package visible ones. Thus at this point we define $R_{\texttt{Class\_ref}}$ as a relation between named items and either external tokens or internal references, subject to coherence constraints.

We must ensure that if a name corresponds to both an external token and to an internal offset, then the token and the offset correspond to the same entity. We ensure this by using the offset function $\texttt{class\_offset} : \texttt{Package\_tok} \times \texttt{Class\_tok} \rightarrow \texttt{Offset}$ which returns the internal offset corresponding to an external token, and then *define* $R_{\texttt{Class\_ref}}$ from this and $R_{\texttt{Ext\_class\_ref}}$. Clearly, therefore, $R_{\texttt{Class\_ref}}$ is not a bijection.

`Class_ref` **:** We define $R_{\texttt{Class\_ref}}$ as an external bijection which respects $R_{\texttt{Ext\_class\_ref}}$, that is, such that

$$\mathit{c\_name} \ R_{\texttt{Class\_ref}} \ (\mathit{p\_tok}, \mathit{c\_tok}) \iff \mathit{c\_name} \ R_{\texttt{Ext\_class\_ref}} \ (\mathit{p\_tok}, \mathit{c\_tok}).$$

Thus $R_{\texttt{Class\_ref}}$ extends $R_{\texttt{Ext\_class\_ref}}$ to internal references.

`Method_info`**:** We only treat certain parts of the method information here:

$$\begin{array}{c} \langle \mathit{flags}, \mathit{sig}, \_, \_, \mathit{maxstack}, \mathit{maxlocals}, \mathit{code}, \_ \rangle \\ R_{\texttt{Method\_info}} \\ \langle \mathit{flags}', \mathit{maxstack}', \mathit{nargs}', \mathit{maxlocals}', \mathit{code}' \rangle \end{array} \iff \begin{array}{l} \mathit{flags} \ R_{\texttt{Method\_flags}} \ \mathit{flags}' \ \wedge \\ \mathit{maxstack} = \mathit{maxstack}' \wedge \\ \mathit{size}(\mathit{sig}) = \mathit{nargs}' \wedge \\ \mathit{maxlocals} = \mathit{maxlocals}' \wedge \\ \mathit{code} \ R_{\texttt{Bytecode}} \ \mathit{code}' \end{array}$$

In the name interpretation, information is grouped by the package and so, for example, $[\![\texttt{Pack\_methods}]\!]_{name} : \texttt{Class\_name} \rightarrow \texttt{Methods\_item}$ is the 'set' of method data for all classes. In the token format the method information is spread between the two components. The coupling relations reflect this: the relation $R_{\texttt{Class}}$ ensures that a named method corresponds to a particular offset, and $R_{\texttt{Pack\_methods}}$ ensures that the entry at this offset is related by $R_{\texttt{Method\_info}}$.

`Pack_methods`: The method item and method component contain the implementations of both static and virtual methods.

$$\begin{array}{ccl} methods\_name & & \forall \langle c\_name, sig\rangle\ R_{\mathtt{Method\_ref}}\ \langle p\_tok, c\_tok, m\_tok\rangle . \\ R_{\mathtt{Pack\_methods}} & \Leftrightarrow & methods\_name(c\_name, sig)\ R_{\mathtt{Method\_info}} \\ method\_comp & & methods\_comp.methods(\mathtt{method\_offset}(p\_tok, c\_tok, m\_tok)) \end{array}$$

`Class`: We define $R_{\mathtt{Class}}$. There are a number of equivalences expressing correctness of the construction of the class component. For the lookup, the significant ones are those between the method tables. These say that if a method is defined in the name format, then it must be defined (and equivalent) in the token format. Since the converse is not required, this means we can copy method tokens from a superclass. Instead, there is a condition saying that if there is a method token, then there must be a corresponding signature in some superclass.

If a method is visible in a class, then there must be an entry in the method table, indicating how to find the method information structure in the appropriate method component. For package visible methods this implies that the method must be in the same package. For public methods, if the two classes are in the same package, then this entry is an offset into the method component of this package. Otherwise, the entry is `0xFFFF`, indicating that we must use the method token to look in another package.

The class component only contains part of the information contained in the class files. The full definition is given in Figure 5. (writing c_name for $cf.Class\_name$ and c_ref for $ci.Class\_ref$): The offset functions link the various relations. We make a global assumption (in fact, local to an environment) of the existence of `class_offset` and `method_offset`.

Equivalence proof: The full proof establishes that the auxiliary functions preserve the appropriate relations [2]. Here, we state the main lemma for the function `lookup` whose type is `Class_ref` × `Method_ref` $\rightarrow$ `Class_ref` × `Bytecode`.

Lemma 1. *If the heap and environment are related in the two formats, then:*

$$[\![lookup]\!]_{name}\ R_{\mathtt{Class\_ref}\times\mathtt{Method\_ref}\rightarrow\mathtt{Class\_ref}\times\mathtt{Bytecode}}\,[\![lookup]\!]_{tok}$$

In order to use the operational semantics with the logical relations approach it is convenient to view the operational semantics as giving an interpretation. We define $[\![code]\!](\langle env, heap, op\_stack, loc\_vars, m\_ref\rangle)$ as the resulting state from the (unique) transition from $\langle code, op\_stack, loc\_vars\rangle$ with environment env and heap $heap$. Thus interpreted bytecode has type `State` $\rightarrow$ `Bytecode` × `State` where `State` is

$$\mathtt{State} = \mathtt{Global\_state} \times \mathtt{Operand\_stack} \times \mathtt{Local\_variables} \times \mathtt{Class\_ref}$$

Now, the following fact is trivial to show: if $R_B = id_B$ for all basic observable types, then $R_\theta = id_\theta$ for all observable θ. In combination with the following theorem, then, this says that if a transformation satisfies certain constraints (formally expressed by saying that it is contained in R) then it is correct, in the sense that no difference can be observed in the two semantics. In particular, we can observe the operand stack (of observable type $\mathtt{Word}^*$) and the local variables (of observable type `Nat` $\rightarrow$ `Word`) so these are identical under the two formats.

$$
\begin{array}{l}
cf : \mathtt{Class\_file}\ R_{\mathtt{Class}}\ ci : \mathtt{Class\_info} \iff \\
\left\{
\begin{array}{l}
cf.Class\_flags\ R_{\mathtt{Class\_flags}}\ ci.Class\_flags\ \wedge \\
cf.Super\ R_{\mathtt{Class\_ref}}\ ci.Super\ \wedge \\
\\
\forall sig \in cf.Methods\_item. \\
public(sig) \Rightarrow \\
\exists m\_tok\ .\ ci.Public\_base \leq m\_tok < ci.Public\_base + ci.Public\_size\ \wedge \\
\langle c\_name, sig \rangle\ R_{\mathtt{Method\_ref}}\ \langle c\_ref, m\_tok \rangle\ \wedge \\
ci.Public\_table[m\_tok - ci.Public\_base] = \mathtt{method\_offset}(c\_ref, m\_tok) \\
\wedge \\
package(sig) \Rightarrow \\
\exists m\_tok\ .\ ci.Package\_base \leq m\_tok\ \&\ 127 < ci.Package\_base + ci.Package\_size\ \wedge \\
\langle c\_name, sig \rangle\ R_{\mathtt{Method\_ref}}\ \langle c\_ref, m\_tok \rangle\ \wedge \\
ci.Package\_table[m\_tok\ \&\ 127 - ci.Package\_base] = \mathtt{method\_offset}(c\_ref, m\_tok) \\
\wedge \\
\forall m\_tok \in ci.Public\_table \cup ci.Package\_table. \exists sig. \exists c'\_name. \\
\langle c'\_name, sig \rangle\ R_{\mathtt{Method\_ref}}\ \langle c\_ref, m\_tok \rangle \wedge c\_name \leq c'\_name\ \wedge \\
public(sig) \Rightarrow [\ (same\_package(c\_name, c'\_name) \iff \\
\qquad\qquad\qquad ci.Public\_table[m\_tok - ci.Public\_base] \neq \mathtt{0xFFFF})]
\end{array}
\right.
\end{array}
$$

Fig. 5. Definition of $R_{\mathtt{Class}}$

Theorem 1. *Assume that* $env_{name}\ R_{\mathtt{Environment}}\ env_{tok}$, $heap_{name}\ R_{\mathtt{Heap}}\ heap_{tok}$, $ls\ R_{\mathtt{Local\_state}}\ ls'$, *and* $code\ R_{\mathtt{Bytecode}}\ code'$. *Then*

$$[\![code]\!]_{name}(env_{name}, heap_{name}, ls)\ R_{\mathtt{Bytecode} \times \mathtt{State}}\ [\![code']\!]_{tok}(env_{tok}, heap_{tok}, ls')$$

8 Conclusion

We have formalised the virtual machines and file formats for Java and Java Card, and the optimisation as a relation between the two. Correctness of this optimisation was expressed in terms of observable equivalence of the operational semantics, and this was deduced from the constraints that define the optimisation. Although the framework we have presented is quite general, the proof is specific to the instantiations of auxiliary functions we chose. It could be argued that we might have proven the equivalence of two incorrect implementations of lookup. The remedy for this would be to specify the functions themselves, and independently prove their correctness. Furthermore, we have made a number of simplifications which could be relaxed. We have used a simple definition of $R_{\mathtt{Bytecode}}$ here, which just accounts for the changing indexes into constant pools (as well as method references in configurations). We have not considered inlining or the specialisation of instructions, however. We expressed equivalence in terms of an identity at observable types but we should also account for the difference in word size, as in [3]. Although the specialisation of instructions could be handled by our technique, the extension is less clear for the more non-local optimisations.

We emphasise that the particular form of operational semantics used here is orthogonal to the rest of the proof. This version suffices for the instructions considered here, but could easily be changed (along with the definition of $R_{\mathtt{Bytecode}}$). The auxiliary functions could be given different definitions; for example, an abstract interpretation or, going in the opposite direction, including error information.

The definitions have been formalised in Coq, and the lemmas verified [9]. The discipline this imposed on the work presented here was very helpful in revealing errors. Even just getting the definitions to type-check uncovered many errors. We take the complexity of the proofs (in Coq) as evidence for the merit in separating the correctness of a particular algorithm from the correctness of the specification. In fact, the operational semantics, correctness of the specification, and development of the algorithm are all largely independent of each other.

As mentioned in the introduction, there are two main steps to showing correctness:

1. Give an abstract characterisation of all possible transformations and show that the abstract properties guarantee correctness.
2. Show that an algorithm implementing such a transformation exists.

We are currently working on a formal development of a tokenisation algorithm using Coq's program extraction mechanism together with constraint-solving tactics.

Acknowledgments

This work was partially supported by the INRIA *Action de recherche coopérative* Java Card

References

1. P. Bertelsen. Semantics of Java byte code. Technical report, Department of Information Technology, Technical University of Denmark, March 1997.
2. Ewen Denney. Correctness of Java Card Tokenisation. Technical Report 1286, Projet Lande, IRISA, 1999. Also appears as INRIA research report 3831.
3. Jean-Louis Lanet and Antoine Requet. Formal proof of smart card applets correctness. In *Third Smart Card Research and Advanced Application Conference (CARDIS'98)*, 1998.
4. T. Lindholm and F. Yelling. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.
5. J. Mitchell. *Foundations for Programming Languages*. Foundations of Computing Series. MIT Press, 1996.
6. Peter D. Mosses. Modularity in structural operational semantics. Extended abstract, November 1998.
7. Cornelia Pusch. Verification of Compiler Correctness for the WAM. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics (TPHOLs'96)*, pages 347–362. Springer-Verlag, 1996.
8. Cornelia Pusch. Formalizing the Java Virtual Machine in Isabelle/HOL. Technical Report TUM-I9816, Institut für Informatik, Technische Universität München, 1998.
9. Gaëlle Segouat. Preuve en Coq d'une mise en oeuvre de Java Card. Master's thesis, Projet Lande, IRISA, 1999.
10. Sun Microsystems. *Java Card 2.0 Language Subset and Virtual Machine Specification*, October 1997. Final Revision.
11. Sun Microsystems. *Java Card 2.1 Virtual Machine Specification*, March 1999. Final Revision 1.0.

Compile-Time Debugging of C Programs Working on Trees

Jacob Elgaard, Anders Møller, and Michael I. Schwartzbach

BRICS, University of Aarhus
{elgaard,amoeller,mis}@brics.dk

Abstract. We exhibit a technique for automatically verifying the safety of simple C programs working on tree-shaped data structures. We do not consider the complete behavior of programs, but only attempt to verify that they respect the shape and integrity of the store. A verified program is guaranteed to preserve the tree-shapes of data structures, to avoid pointer errors such as NULL dereferences, leaking memory, and dangling references, and furthermore to satisfy assertions specified in a specialized store logic.

A program is transformed into a single formula in WSRT, an extension of WS2S that is decided by the MONA tool. This technique is complete for loop-free code, but for loops and recursive functions we rely on Hoare-style invariants. A default well-formedness invariant is supplied and can be strengthened as needed by programmer annotations. If a program fails to verify, a counterexample in the form of an initial store that leads to an error is automatically generated.

This extends previous work that uses a similar technique to verify a simpler syntax manipulating only list structures. In that case, programs are translated into WS1S formulas. A naive generalization to recursive data-types determines an encoding in WS2S that leads to infeasible computations. To obtain a working tool, we have extended MONA to directly support recursive structures using an encoding that provides a necessary state-space factorization. This extension of MONA defines the new WSRT logic together with its decision procedure.

1 Introduction

Catching pointer errors in programs is a difficult task that has inspired many assisting tools. Traditionally, these come in three flavors. First, tools such a Purify [3] and Insure++ [17] instrument the generated code to monitor the runtime behavior thus indicating errors and their sources. Second, traditional compiler technologies such as program slicing [21], pointer analysis [7], and shape analysis [19] are used in tools like CodeSurfer [8] and Aspect [10] that conservatively detect known causes of errors. Third, full-scale program verification is attempted by tools like LCLint [6] and ESC [5], which capture runtime behavior as formulas and then appeal to general theorem provers.

All three approaches lead to tools that are either incomplete or unsound (or both), even for straight-line code. In practice, this may be perfectly acceptable if a significant number of real errors are caught.

G. Smolka (Ed.): ESOP/ETAPS 2000, LNCS 1782, pp. 119–134, 2000.

In previous work [11], we suggest a different balance point by using a less expressive program logic for which Hoare triples on loop-free code is decidable when integer arithmetic is ignored. That work is restricted by allowing only a while-language working on linear lists. In the present paper we extend our approach by allowing recursive functions working on recursive data-types. This generalization is conceptually simple but technically challenging, since programs must now be encoded in WS2S rather than the simpler WS1S. Decision procedures for both logics are provided by the MONA tool [13, 18] on which we rely, but a naive generalization of the previous encoding leads to infeasible computations. We have responded by extending MONA to directly support a logic of recursive data-types, which we call WSRT. This logic is encoded in WS2S in a manner that exploits the internal representation of MONA automata to obtain a much needed state-space factorization.

Our resulting tool catches all pointer errors, including NULL dereferences, leaking memory, and dangling references. It can also verify assertions provided by the programmer in a special store logic. The tool is sound and complete for loop-free code including if-statements with restricted conditions: it will reject exactly the code that may cause errors or violate assertions when executed in some initial store. For while-loops or functions, the tool relies on annotations in the form of invariants and pre- and post-conditions. In this general case, our tool is sound but incomplete: safe programs exist that cannot be verified regardless of the annotations provided. In practical terms, we provide default annotations that in many cases enable verification.

Our implementation is reasonably efficient, but can only handle programs of moderate sizes, such as individual operations of data-types. If a program fails to verify, a counterexample is provided in the form of an initial store leading to an error. A special simulator is supplied that can trace the execution of a program and provide graphical snapshots of the store. Thus, a reasonable form of compile-time debugging is made available. While we do not detect all program errors, the verification provided serves as a finely masked filter for most bugs.

As an example, consider the following recursive data-type of binary trees with red, green, or blue nodes:

```
struct RGB {
  enum {red,green,blue} color;
  struct RGB *left;
  struct RGB *right;
};
```

The following non-trivial application collects all green leaves into a right-linear tree and changes all the blue nodes to become red:

```
/**data**/ struct RGB *tree;
/**data**/ struct RGB *greens;

enum bool {false,true};

enum bool greenleaf(struct RGB *t) {
  if (t==0) return false;
```

```
    if (t->color!=green) return false;
    if (t->left!=0 || t->right!=0) return false;
    return true;
  }

  void traverse(struct RGB *t) {
    struct RGB *x;
    if (t!=0) {
      if (t->color==blue) t->color = red;
      if (greenleaf(t->left)==true /**keep: t!=0 **/) {
        t->left->right = greens;
        greens = t->left;
        t->left=0;
      }
      if (greenleaf(t->right)==true /**keep: t!=0 **/) {
        t->right->right = greens;
        greens = t->right;
        t->right=0;
      }
      traverse(t->left); /**keep: t!=0 **/
      traverse(t->right); /**keep: t!=0 **/
    }
  }

  /**pre: greens==0 **/
  main()  { traverse(tree); }
```

The special comments are assertions that the programmer must insert to specify the intended model (`/**data**/`), restrict the set of stores under consideration (`/**pre**/`), or aid the verifier (`/**keep**/`). They are explained further in Section 2.4.

Without additional annotations, our tool can verify this program (in 33 seconds on a 266MHz Pentium II PC with 128 MB RAM). This means that no pointer errors occur during execution from any initial store. Furthermore, both `tree` and `greens` are known to remain well-formed trees. Using the assertion:

```
  all p: greens(->left + ->right)*==p => (p!=0 => p->color==green)
```

we can verify (in 74 seconds) that `greens` after execution contains only green nodes. That `greens` is right-linear is expressed through the assertion:

```
  all p: greens(->left + ->right)*==p => (p!=0 => p->left==0)
```

In contrast, if we assert that `greens` ends up empty, the tool responds with a minimal counterexample in the form of an initial store in which `tree` contains a green leaf.

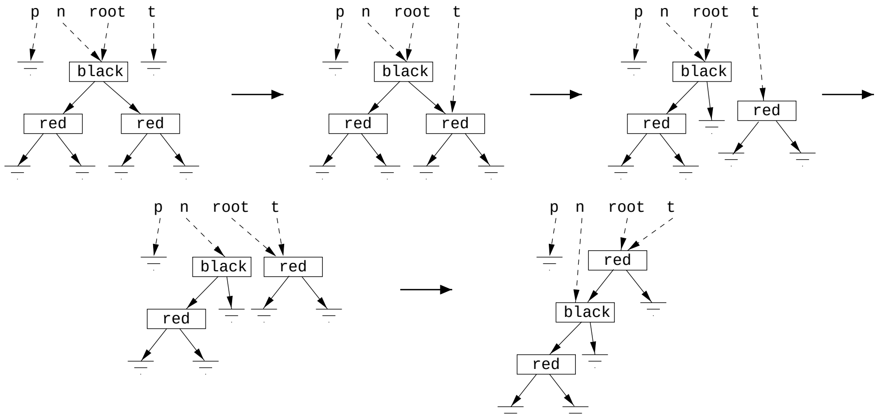
An example of the simulator used in conjunction with counterexamples comes from the following fragment of an implementation of red-black search trees. Consider the following program, which performs a left rotation of a node `n` with parent `p` in such a tree:

```
struct Node {
  enum {red, black} color;
  struct Node *left;
  struct Node *right;
};

/**data**/ struct Node *root;

/**pre: n!=0 & n->right!=0 &
        (p!=0 => (p->left==n | p->right==n)) &
        (p==0 => n==root) **/
void left_rotate(struct Node *n, struct Node *p) {
  struct Node *t;
  t = n->right;
  n->right = t->left;
  if (n==root) root = t;
  else if (p->left==n) p->left = t;
  else p->right = t;
  t->left = n;
}
```

In our assertion language, we cannot express the part of the red-black data-type invariant that each path from the root to a leaf must contain the same number of black nodes; however we can capture the part that the root is black and that a red node cannot have red children:

```
root->color==black &
all p: p->color==red =>
       (p->left->color!=red & p->right->color!=red)
```

If we add the above assertion as a data-type invariant, we are (in 18 seconds) given a counterexample. If we apply the simulator, we see the following example run, which shows that we have forgotten to consider that the root may become red (in which case we should add a line of code coloring it black):



Such detailed feedback at compile-time is clearly a useful debugging tool.

2 The Language

The language in consideration is a simple yet non-trivial subset of C. It allows declaration of tree-shaped recursively typed data structures and recursive imperative functions operating on the trees. The subset is chosen such that the verification we intend to perform becomes decidable. Thus, for instance, integer arithmetic is omitted from the language; only finite enumeration types can be expressed. Also, to smoothen presentation, many other C constructs have been omitted although some of them easily could be included directly, and other by approximating their behavior.

We begin by defining the core language. After that, we describe how programs can be annotated with formulas expressing additional requirements for correctness.

2.1 The C Subset

The abstract syntax of the C subset is defined using EBNF notation, where furthermore ⊛ is used to denote comma-separated lists with zero or more elements. The semantics of the language is as known from C.

A program consists of declarations of structures, enumerations, variables, and functions:

```
program  →  ( struct | enum | var | function )*
```

A structure contains an enumeration denoting its value and a union of structures containing pointers to its child structures. An enumeration is a list of identifiers:

```
struct  →  struct id {
              enum id id;
              union {
                ( struct {
                  ( struct id * id; )*
                } id; )*
              } id;
            };
enum    →  enum id { id+ };
```

The enumeration values denote the *kind* of the structure, and the kind determines which is the *active* union member. The association between enumeration values and union members is based on their indices in the two lists. Such data structures are typical in real-world C programs and exactly define recursive data-types. One goal of our verification is to ensure that only active union members are accessed.

For abbreviation we allow declarations of structures and enumerations to be inlined. Also, we allow (`struct` *id* `*` *id*;)* in place of `union` {...}, implicitly meaning that all union members are identical. A variable is either a pointer to a structure or an enumeration:

```
var   →  type id;
type  →  struct id * | enum id
```

A function can contain variable declarations and statements:

$$\begin{array}{lcl} \mathit{function} & \rightarrow & (\ \texttt{void}\ \mid\ \mathit{type}\ )\ \mathit{id}\texttt{(}\ (\ \mathit{type}\ \mathit{id}\ )^{\circledast}\ \texttt{)}\ \texttt{\{} \\ & & \quad \mathit{var}^{*}\ \mathit{stm}^{?} \\ & & \quad (\ \texttt{return}\ \mathit{rvalue}\texttt{;}\ )^{?} \\ & & \texttt{\}} \end{array}$$

A statement is a sequence, an assignment, a function call, a conditional statement, a `while`-loop, or a memory deallocation:

$$\begin{array}{lcl} \mathit{stm} & \rightarrow & \mathit{stm}\ \mathit{stm}\ \mid \\ & & \mathit{lvalue}\ \texttt{=}\ \mathit{rvalue}\texttt{;}\ \mid \\ & & \mathit{id}\texttt{(}\ (\ \mathit{rvalue}\ )^{\circledast}\ \texttt{);}\ \mid \\ & & \texttt{if (}\ \mathit{cond}\ \texttt{)}\ \mathit{stm}\ (\ \texttt{else}\ \mathit{stm}\ )^{?}\ \mid \\ & & \texttt{while (}\ \mathit{cond}\ \texttt{)}\ \mathit{stm}\ \mid \\ & & \texttt{free(}\ \mathit{lvalue}\ \texttt{);} \end{array}$$

A condition is a boolean expression evaluating to either true or false; the expression `?` represents non-deterministic choice and can be used in place of those C expressions that are omitted from our subset language:

$$\mathit{cond} \quad \rightarrow \quad \mathit{cond}\ \texttt{\&}\ \mathit{cond}\ \mid\ \mathit{cond}\ \texttt{|}\ \mathit{cond}\ \mid\ \texttt{!}\ \mathit{cond}\ \mid\ \mathit{rvalue}\ \texttt{==}\ \mathit{rvalue}\ \mid\ \texttt{?}$$

An *lvalue* is an expression designating an enumeration variable or a pointer variable. An *rvalue* is an expression evaluating to an enumeration value or to a pointer to a structure. The constant `0` is the NULL pointer, `malloc` allocates memory on the heap, and *id*`(...)` is a function call:

$$\begin{array}{lcl} \mathit{lvalue} & \rightarrow & \mathit{id}\ (\ \texttt{->}\ \mathit{id}\ (\ \texttt{.}\ \mathit{id}\ )^{?}\ )^{*} \\ \mathit{rvalue} & \rightarrow & \mathit{lvalue}\ \mid\ \texttt{0}\ \mid\ \texttt{malloc(sizeof(}\ \mathit{id}\ \texttt{))}\ \mid\ \mathit{id}\texttt{(}\ \mathit{rvalue}^{\circledast}\ \texttt{)} \end{array}$$

The nonterminal *id* represents identifiers.

The presentation of our verification technique is based on C for familiarity reasons only—no intrinsic C constructs are utilized.

2.2 Modeling the Store

During execution of a program, structures located in the heap are allocated and freed, and field variables and local variables are assigned values. The state of an execution can be described by a model of the heap and the local variables, called the *store*.

A store is modeled as a finite graph, consisting of a set of *cells* representing structures, a distinguished *NULL cell*, a set of *program variables*, and *pointers* from cells or program variables to cells. Each cell is labeled with a *value* taken from the enumerations occurring in the program. Furthermore, each cell can have a *free* mark, meaning that it is currently not allocated.

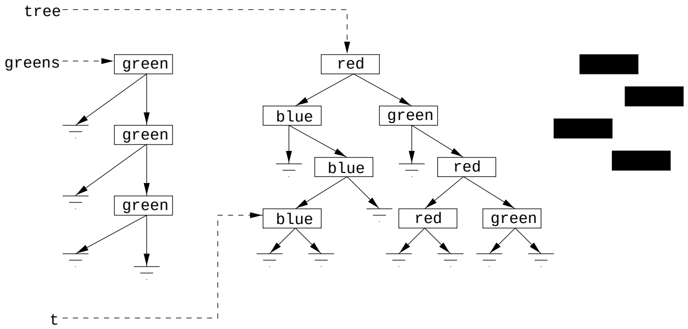
Program variables are those that are declared in the program either globally or inside functions. To enable the verification, we need to classify these variables as either *data* or *pointer* variables. A variable is classified as a data variable by prefixing its declaration in the program with the special comment `/**data**/`; otherwise, it is considered a pointer variable.

A store is *well-formed* if it satisfies the following properties:

- the cells and pointers form disjoint tree structures (the NULL cell may be shared, though);
- each data variable points either to the root of a tree or to the NULL cell;
- each pointer variable points to any cell (including the NULL cell);
- a cell is marked as free if and only if it is not reachable from a program variable; and
- the type declarations are respected—this includes the requirement that a cell representing a structure has an outgoing pointer for each structure pointer declared in its active union member.

With the techniques described in the remainder of this paper, it is possible to automatically verify whether well-formedness is preserved by all functions in a given program. Furthermore, additional user defined properties expressed in the logic presented in Section 2.3 can be verified.

The following illustrates an example of a well-formed store containing some `RGB`-trees as described in Section 1. Tree edges are solid lines whereas the values of pointer variables are dashed lines; free cells are solid black:



2.3 Store Logic

Properties of stores can conveniently be stated using logic. The declarative and succinct nature of logic often allows simple specifications of complex requirements. The logic presented here is essentially a first-order logic on finite tree structures [20]. It has the important characteristic of being decidable, which we will exploit for the program verification.

A formula ϕ in the store logic is built from boolean connectives, first-order quantifiers, and basic propositions. A term t denotes either an enumeration value or a pointer to a cell in the store. A path set P represents a set of paths, where a path is a sequence of pointer dereferences and union member selections ending in either a pointer or an enumeration field. The signature of the logic consists of dereference functions, path relations, and the relations `free` and `root`:

$$\begin{array}{lll} \phi & \rightarrow & \texttt{!}\ \phi \mid \phi\ \texttt{\&}\ \phi \mid \phi\ \texttt{|}\ \phi \mid \phi\ \texttt{=>}\ \phi \mid \phi\ \texttt{<=>}\ \phi \mid \\ & & \texttt{ex}\ id\ \texttt{:}\ \phi \mid \texttt{all}\ id\ \texttt{:}\ \phi \mid \texttt{true} \mid \texttt{false} \mid \\ & & id\ \texttt{(}\ P\ \texttt{)}^{?}\ \texttt{==}\ t \mid \texttt{free(}\ t\ \texttt{)} \mid \texttt{root(}\ t\ \texttt{)} \end{array}$$

A path relation, $id\ P$ `==` t, compares either enumeration values or cell pointers. The identifier id may be either a bound quantified variable, a program variable, or an enumeration value, and t is a term.

If both id and t denote cell pointers, a path relation is true for a given store if there is a path in P from the cell denoted by id to the cell denoted by t in the store. If P is omitted, the relation is true if id and t denote the same cell.

If id denotes a cell pointer and t is an enumeration value, a path relation is true for a given store if there is a path satisfying P from the cell denoted by id to an enumeration field with the value t in the store.

The relation `free(`t`)` is true in a given store if the cell denoted by t is marked as not allocated in the store. The relation `root(`t`)` is true if t denotes the root of some tree.

A term is a sequence of applications of the dereference function and union member selections or the constant `0` representing the special NULL cell:

$$t \quad \rightarrow \quad id\ (\ \texttt{->}\ id\ (\ \texttt{.}\ id\ )^{?}\ )^{*}\ \mid\ \texttt{0}$$

A path set is a regular expression:

$$P \quad \rightarrow \quad \texttt{->}\ id\ (\ \texttt{.}\ id\ )^{?}\ \mid\ P\ \texttt{+}\ P\ \mid\ P\ P\ \mid\ P\ \texttt{*}$$

The path set defined by `->`id_1`.`id_2 consists of a single dereference of id_1 and subsequent selection of the member id_2. The expressions P `+` P, $P\ P$, and P `*` respectively denote union, concatenation, and Kleene star.

2.4 Program Annotations and Hoare Triples

The verification technique is based on Hoare triples [9], that is, constructs of the form $\{\phi_1\}stm\{\phi_2\}$. The meaning of this triple is that executing the statement stm in a store satisfying the pre-condition ϕ_1 always results in a store satisfying the post-condition ϕ_2, provided that the statement terminates. Well-formedness is always implicitly included in both ϕ_1 and ϕ_2. We can only directly decide such triples for loop-free code. Programs containing loops—either as `while`-loops or as function calls—must be split into loop-free fragments.

A program can be annotated with formulas expressing requirements for correctness using a family of designated comments. These annotations are also used to split the program into a set of Hoare triples that subsequently can be verified separately.

`/**pre:` ϕ `**/` and `/**post:` ϕ `**/` may be placed between the signature and the body of a function. The `pre` formula expresses a property that the verifier may assume initially holds when the function is executed. The `post` formula expresses a property intended to hold after execution of the function. The states before and after execution may be related using otherwise unused variables.

`/**inv:` ϕ `**/` may be placed between the condition and the body of a `while`-loop. It expresses an invariant property that must hold before execution of the loop and after each iteration. It splits the code into three parts: the statements preceding the `while`-loop, its body, and the statements following it.

`/**keep:` ϕ `**/` may be placed immediately after a function call. It expresses a property that must hold both before and after the call. It splits the code into two parts: the statements before and after the call. The `keep` formulas can specify invariants for recursive function calls just as `inv` formulas can specify invariants for `while`-loops.

`/**assert:` ϕ `**/` may be placed between statements. It splits the statement sequence, such that a Hoare triple is divided into two smaller triples, where the post-condition of the first and the pre-condition of the second both are ϕ. This allows modular analysis to be performed. The variant `/**assert:` ϕ `assume:` ϕ `**/` allows the post-condition and the pre-condition to be different, and thereby to weaken the verification requirements. This is needed whenever a sufficiently strong property either cannot be expressed or requires infeasible computations.

`/**check:` ϕ `**/` *stm* informally corresponds to "`if (!`ϕ`) fail; else` *stm*", where `fail` is some statement that fails to verify. This can be used to check that a certain property holds without creating two Hoare triples incurring a potential loss of information.

Whenever a pre- or post-condition, an invariant, or a keep-formula is omitted, the default formula `true` is implicitly inserted. Actually, many interesting properties can be verified with just these defaults. As an example, the program:

```
/**data**/ struct RGB *x;
struct RGB *p;
struct RGB *q;

p = x;
q = 0;
while (p!=0 & q==0) /**inv: q!=0 => q->color==red **/ {
  if (p->color==red) q = p;
  else if (p->color==green) p = p->left;
  else /**assert: p->color==blue **/ p = p->right;
}
```

yields the following set of Hoare triples and logical implications to be checked:

```
{ true } p = x; q = 0; { I }
( I & !B ) => true
{ I & B & B1 } q = p; { I }
{ I & B & !B1 & B2 } p = p->left; { I }
( I & B & !B1 & !B2 ) => ( p->color==blue )
{ I & B & !B1 & !B2 & p->color==blue } p = p->right; { I }
```

where `B` is the condition of the `while`-loop, `I` is the invariant, `B1` is the condition of the outer `if`-statement and `B2` that of the inner `if`-statement. Note that the generated Hoare triples are completely independent of each other—when a triple is divided into two smaller triples, no information obtained from analyzing the first triple is used when analyzing the second.

3 Deciding Hoare Triples

The generated Hoare triples and logical implications—both the formula parts and the program parts—can be encoded in the logic WS2S which is known to be decidable. This encoding method follows directly from [11] by generalizing from list structures to tree structures in the style of [16]. The MONA tool provides an implementation of a decision procedure for WS2S, so in principle making a decision procedure for the present language requires no new ideas.

As we show in the following, this method will however lead to infeasible computations making it useless in practice. The solution is to exploit the full power of the MONA tool: usually, WS2S is decided using a correspondence with ordinary tree automata—MONA uses a representation called *guided tree automata*, which when used properly can be exponentially more efficient than ordinary tree automata. However, such a gain requires a non-obvious encoding.

We will not describe how plain MONA code directly can be generated from the Hoare triples and logical implications. Instead we introduce a logic called *WSRT*, *weak monadic second-order logic with recursive types*, which separates the encoding into two parts: the Hoare triples and logical implications are first encoded in WSRT, and then WSRT is translated into basic MONA code. This has two benefits: WSRT provides a higher level of abstraction for the encoding task, and, as a by-product, we get an efficient implementation of a general tree logic which can be applied in many other situations where WS2S and ordinary tree automata have so far been used.

3.1 Weak Monadic Second-Order Logic with Recursive Types

A *recursive type* is a set of recursive equations of the form:

$$T_i = v_1(c_{1,1} : T_{j_{1,1}}, \ldots, c_{1,m_1} : T_{j_{1,m_1}}), \ldots, v_n(c_{n,1} : T_{j_{n,1}}, \ldots, c_{n,m_n} : T_{j_{n,m_n}})$$

Each T denotes the name of a type, each v is called a *variant*, and each c is called a *component*. A tree conforms to a recursive type T if its root is labeled with a variant v from T and it has a successor for each component in v such that the successor conforms to the type of that component. Note that types defined by `struct`s in the language in Section 2.1 exactly correspond to such recursive types.

The logic WSRT is a weak monadic second-order logic. Formulas are interpreted relative to a set of trees conforming to recursive types. Each node is labeled with a variant from a recursive type. A tree variable denotes a tree conforming to a fixed recursive type. A first-order variable denotes a single node. A second-order variable denotes a finite set of nodes.

A formula is built from the usual boolean connectives, first-order and weak monadic second-order quantifiers, and the special WSRT basic formulas:

type(t, T) which is true iff the the first-order term t denotes a node which is labeled with some variant from the type T; and

variant(t, x, T, v) which is true iff the tree denoted by the tree variable x at the position denoted by t is labeled with the T variant v.

Second-order terms are built from second-order variables and the set operations union, intersection and difference. First-order terms are built from first-order variables and the special WSRT functions:

$\mathit{tree\_root}(x)$ which evaluates to the root of the tree denoted by x; and
$\mathit{succ}(t, T, v, c)$ which, provided that the first-order term t denotes a node of the T variant v, evaluates to its c component.

This logic corresponds to the core of the FIDO language [16] and is also reminiscent of the LISA language [1]. It can be reduced to WS2S and thus provides no more expressive power, but we will show that a significantly more efficient decision procedure exists if we bypass WS2S.

3.2 Encoding Stores and Formulas in WSRT

The idea behind the decision procedure for Hoare triples is to encode well-formed stores as trees. The effect of executing a loop-free program fragment is then in a finite number of steps to transform one tree into another. WSRT can conveniently be used to express regular sets of finite trees conforming to recursive types, which turns out to be exactly what we need to encode pre- and post-conditions and effects of execution.

We begin by making some observations that simplify the encoding task. First, note that NULL pointers can be represented by adding a “NULL kind” with no successors to all structures. Second, note that memory allocation issues can be represented by having a “free list” for each `struct`, just as in [11]. We can now represent a well-formed store by a set of WSRT variables:

- each data variable is represented by a WSRT tree variable with the same recursive type, where we use the fact that the types defined by `struct`s exactly correspond to the WSRT notion of recursive types; and
- each pointer variable in the program is represented by a WSRT first-order variable.

For each program point, a set of WSRT predicates called *store predicates* is used to express the possible stores:

- for each data variable d in the program, the predicate $\mathit{root}_d(t)$ is true whenever the first-order term t denotes the root of d;
- for each pointer variable p, the predicate $\mathit{pos}_p(t)$ is true whenever t and p denote the same position;
- for each pointer field f occurring in a union u in some structure s, the predicate $\mathit{succ}_{f,u,s}(t_1, t_2)$ is true whenever the first-order term t_1 points to a cell of type s having the value u, and the f component of this cell points to the same node as the first-order term t_2;
- for each possible enumeration value e, the predicate $\mathit{kind}_e(t)$ is true whenever t denotes a cell with value e; and
- to encode allocation status, the predicate $\mathit{free}_s(t)$ is true whenever t denotes a non-allocated cell.

A set of store predicates called the *initial store predicates* defining a mapping of the heap into WSRT trees can easily be expressed in the WSRT logic. For instance, the initial store predicates *root*, *succ*, and *kind* simply coincide with the corresponding basic WSRT constructs.

Based on a set of store predicates, the well-formedness property and all store-logic formulas can be encoded as other predicates. For well-formedness, the requirements of the recursive types are expressed using the *root*, *kind*, and *succ* predicates, and the requirement that all data structures are disjoint trees is a simple reachability property. For store-logic formulas, the construction is inductive: boolean connectives and quantifiers are directly translated into WSRT; terms are expressed using the store predicates *root*, *kind*, and *succ*; and the basic formulas `free(`t`)` and `root(`t`)` can be expressed using the store predicates *free* and *root*. Only the regular path sets are non-trivial; they are expressed in WSRT using the method from [14] (where path sets are called "routing expressions"). Note that even though the logic in Section 2.3 is a first-order logic, we also need the weak monadic second-order fragment of WSRT to express well-formedness and path sets.

3.3 Predicate Transformation

When the program has been broken into loop-free fragments, the Hoare triples are decided using the transduction technique introduced in [15]. In this technique, the effect of executing a loop-free program fragment is simulated, step by step, by transforming store predicates accordingly, as described in the following.

Since the pre-condition of a Hoare triple always implicitly includes the well-formedness criteria, we encode the set of *pre-stores* as the conjunction of well-formedness and the pre-condition, both encoded using the initial store predicates, and we initiate the transduction with the initial store predicates. For each step, a new set of store predicates is defined representing the possible stores after executing that step. This predicate transformation is performed using the same ideas as in [11], so we omit the details.

When all steps in this way have been simulated, we have a set of *final* store predicates which exactly represents the changes made by the program fragment. We now encode the set of *post-stores* as the conjunction of well-formedness and the post-condition, both encoded using the final store predicates. It can be shown that the resulting predicate representing the post-stores coincides with the weakest precondition of the code and the post-condition. The Hoare triple is satisfied if and only if the encoding of the pre-stores implies the encoding of the post-stores.

Our technique is sound: if verification succeeds, the program is guaranteed to contain no errors. For loop-free Hoare triples, it is also complete. That is, every effect on the store can be expressed in the store logic, and this logic is decidable. In general, no approximation takes place—all effects of execution are simulated precisely. Nevertheless, since not all true properties of a program containing loops can be expressed in the logic, the technique is in general not complete for whole programs.

4 Deciding WSRT

As mentioned, there is a simple reduction from WSRT to WS2S, and WS2S can be decided using a well-known correspondence between WS2S formulas and ordinary tree automata. The resulting so-called *naive* decision procedure for WSRT is essentially the same as the ones used in FIDO and LISA and as the "conventional encoding of parse trees" in [4]. The naive decision procedure along with its deficiencies is described in Section 4.1. In Section 4.2 we show an efficient decision procedure based on the more sophisticated notion of guided tree automata.

4.1 The Naive Decision Procedure

WS2S, the weak monadic second-order logic of two successors, is a logic that is interpreted relative to a binary tree. A first-order variable denotes a single node in the tree, and a second-order variable denotes a finite set of nodes. For a full definition of WS2S, see [20] or [13].

The decision procedure implemented in MONA inductively constructs a tree automaton for each sub-formula, such that the set of trees accepted by the automaton is the set of interpretations that satisfy the sub-formula. This decision procedure not only determines validity of formulas; it also allows construction of counterexamples whenever a formula is not valid.

Note that the logic WSnS, where each node has n successors instead of just two, easily can be encoded in WS2S by replacing each node with a small tree with n leaves. The idea in the encoding is to have a one-to-one mapping from nodes in a WSRT tree to nodes in a WSnS tree, where we choose n as the maximal fanout of all recursive types.

Each WSRT tree variable x is now represented by b second-order variables $v_1, \ldots, v_b$ where b is the number of bits needed to encode the possible type variants. For each node in the n-ary tree, membership in $v_1 \ldots v_b$ represents some binary encoding of the label of the corresponding node in the x tree.

Using this representation, all the basic WSRT formulas and functions can now easily be expressed in WSnS. We omit the details. For practical applications, this method leads to intractable computations requiring prohibitive amounts of time and space. Even a basic concept such as *type well-formedness* yields immense automata. Type well-formedness is the property that the values of a given set of WS2S variables do represent a tree of a particular recursive type.

This problem can be explained as follows. The WS2S encoding is essentially the same as the "conventional encoding of parse trees" in [4], and type well-formedness corresponds to grammar well-formedness. In that paper, it is shown that the number of states in the automaton corresponding to the grammar well-formedness predicate is linear in the size of the grammar, which in our case corresponds to the recursive types. As argued e.g. in [12], tree automata are at least quadratically more difficult to work with than string automata, since the transition tables are two-dimensional as opposed to one-dimensional. This inevitably causes a blowup in time and space requirements for the whole decision procedure.

By this argument, it would be pointless making an implementation based on the described encoding. This claim is supported by experiments with some very simple examples; in each case, we experienced prohibitive time and space requirements.

4.2 A Decision Procedure using Guided Tree Automata

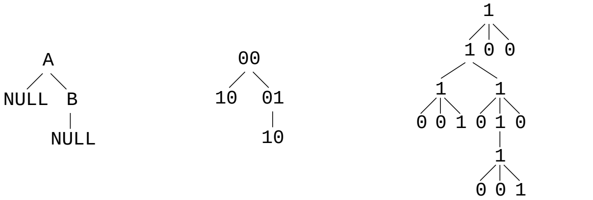
The MONA implementation of WS2S provides an opportunity to factorize the state-space and hence make implementation feasible. To exploit this we must, however, change the encoding of WSRT trees, as described in the following.

The notion of guided tree automata (GTA) was introduced in [2] to combat state-space explosions and is now fully implemented in MONA [13]. A GTA is a tree automaton equipped with separate state spaces that—independently of the labeling of the tree—are assigned to the tree nodes by a top-down automaton, called the *guide*. The secret behind a good factorization is to create the right guide.

A recursive type is essentially also a top-down automaton, so the idea is to derive a guide from the recursive types. This is however not possible with the naive encoding, since the type of a WSnS node depends on the actual value of its parent node.

Instead of using the one-to-one mapping from WSRT tree nodes to WSnS tree nodes labeled with type variants, we represent a WSRT tree entirely by the *shape* of a WSnS tree, similarly to the "shape encoding" in [4]. Each node in the WSRT tree is represented by a WSnS node with a successor node for each variant, and each of these nodes have themselves a successor for each component in the variant. A WSRT tree is then represented by a *single* second-order WSnS variable whose value indicates the active variants.

The following illustrates an example of a tree conforming to the recursive type `Tree=A(left:Tree,right:Tree),B(next:Tree),NULL` and its encodings:



(a) a tree (b) its naive encoding (c) its shape encoding

This encoding has the desired property that a WSnS tree position always is assigned the same type, independently of the tree values, so a GTA guide can directly be derived from the types. This guide factorizes the state space such that all variants and components in the recursive types have their own separate state spaces. Furthermore, the intermediate nodes caused by the WSnS to WS2S transformation can now also be given separate state spaces, causing yet a degree of factorization.

One consequence is that type well-formedness now can be represented by a GTA with a constant number of states in each state space. The size of this automaton is thus reduced from quadratic to linear in the size of the type. Similar improvements are observed for other predicates.

With these obstacles removed, implementation becomes feasible with typical data-type operations verified in seconds. In fact, for the linear sub-language, our new decision procedure is almost as fast as the previous WS1S implementation; for example, the programs `reverse` and `zip` from [11] are now verified in 2.3 and 29 seconds instead of the previous times of 2.7 and 10 seconds (all using the newest version of MONA). This is remarkable, since our decision procedure suffers a quadratic penalty from using tree automata rather than string automata.

5 Conclusion

By introducing the WSRT logic and exploiting novel features of the MONA implementation, we have built a tool that catches pointer errors in programs working on recursive data structures. Together with assisting tools for extracting counterexamples and graphical program simulations, this forms the basis for a compile-time debugger that is sound and furthermore complete for loop-free code. The inherent non-elementary lower bound of WSnS will always limit its applicability, but we have shown that it handles some realistic examples.

Among the possible extensions or variations of the technique are allowing parent and root pointers in all structures, following the ideas from [14], and switching to a finer store granularity to permit casts and pointer arithmetic. A future implementation will test these ideas. Also, it would be interesting to perform a more detailed comparison of the technique presented here with pointer analysis and shape analysis techniques.

References

[1] Abdelwaheb Ayari, David Basin, and Andreas Podelski. LISA: A specification language based on WS2S. In *Proceedings of CSL'97*. BRICS, 1997.

[2] Morten Biehl, Nils Klarlund, and Theis Rauhe. Algorithms for guided tree automata. In *First International Workshop on Implementing Automata, WIA'96*, volume 1260 of *LNCS*. Springer Verlag, 1996.

[3] Rational Software Corporation. Purify. URL: `http://www.rational.com/`.

[4] Niels Damgaard, Nils Klarlund, and Michael I. Schwartzbach. YakYak: Parsing with logical side constraints. In *Proceedings of DLT'99*, 1999.

[5] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. URL: `http://www.research.digital.com/SRC/esc/Esc.html`.

[6] David Evans. LCLint user's guide. URL: `http://www.sds.lcs.mit.edu/lclint/guide/`.

[7] Rakesh Ghiya and Laurie J. Hendren. Putting pointer analysis to work. In *Proceedings of POPL'98*, 1998.

[8] GrammaTech Inc. CodeSurfer user guide and reference manual. URL: `http://www.grammatech.com/papers/`.

[9] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[10] D. Jackson. Aspect: an economical bug-detector. In *Proceedings of 13th International Conference on Software Engineering*, 1994.

[11] Jacob L. Jensen, Michael E. Jørgensen, Nils Klarlund, and Michael I. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. In *PLDI '97*, 1997.

[12] Nils Klarlund. Mona & Fido: The logic-automaton connection in practice. In *Computer Science Logic, CSL '97*, LNCS, 1998.

[13] Nils Klarlund and Anders Møller. *MONA Version 1.3 User Manual.* BRICS Notes Series NS-98-3 (2.revision), Department of Computer Science, University of Aarhus, October 1998. URL: `http://www.brics.dk/mona/manual.html`.

[14] Nils Klarlund and Michael I. Schwartzbach. Graph types. In *Proc. 20th Symp. on Princ. of Prog. Lang.*, pages 196–205. ACM, 1993.

[15] Nils Klarlund and Michael I. Schwartzbach. Graphs and decidable transductions based on edge constraints. In *Proc. CAAP' 94, LNCS 787*, 1994.

[16] Nils Klarlund and Michael I. Schwartzbach. A domain-specific language for regular sets of strings and trees. *IEEE Transactions on Software Engineering*, 25(3), 1997.

[17] Adam Kolawa and Arthur Hicken. Insure++: A tool to support total quality software. URL: `http://www.parasoft.com/products/insure/papers/tech.htm`.

[18] Anders Møller. MONA project homepage. URL: `http://www.brics.dk/mona/`.

[19] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *Transactions on Programming Languages and Systems*, 20(1):1–50, January 1998.

[20] Wolfgang Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 133–191. MIT Press/Elsevier, 1990.

[21] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

A Calculus for Compiling and Linking Classes

Kathleen Fisher[1], John Reppy[2], and Jon G. Riecke[2]

[1] AT&T Labs — Research
180 Park Avenue, Florham Park, NJ 07932 USA
kfisher@research.att.com
[2] Bell Laboratories, Lucent Technologies
700 Mountain Avenue, Murray Hill, NJ 07974 USA
{jhr,riecke}@research.bell-labs.com

Abstract. We describe $\lambda\mathit{ink}\varsigma$ (pronounced "links"), a low-level calculus designed to serve as the basis for an intermediate representation in compilers for class-based object-oriented languages. The primitives in $\lambda\mathit{ink}\varsigma$ can express a wide range of class-based object-oriented language features, including various forms of inheritance, method override, and method dispatch. In particular, $\lambda\mathit{ink}\varsigma$ can model the object-oriented features of Moby, OCaml, and Loom, where subclasses may be derived from unknown base classes. $\lambda\mathit{ink}\varsigma$ can also serve as the intermediate representation for more conventional class mechanisms, such as Java's. In this paper, we formally describe $\lambda\mathit{ink}\varsigma$, give examples of its use, and discuss how standard compiler transformations can be used to optimize programs in the $\lambda\mathit{ink}\varsigma$ representation.

1 Introduction

Class-based object-oriented languages provide mechanisms for factoring code into a hierarchy of classes. For example, the implementation of a text window may be split into a base class that implements windows and a subclass that supports drawing text. Since these classes may be defined in separate compilation units, compilers for such languages need an intermediate representation (IR) that allows them to represent code fragments (*e.g.*, the code for each class) and to generate linkage information to assemble the fragments. For languages with manifest class hierarchies (*i.e.*, languages where subclass compilation requires the superclass representation, as is the case in C++ [Str97] and Java [AG98]), representing code fragments and linkage information is straightforward. But for languages that allow classes as module parameters, such as Moby [FR99a] and OCaml [RV98,Ler98], or languages that have classes as first-class values, such as Loom [BFP97], the design of an IR becomes trickier (Section 2 illustrates the complications).

We are interested in a compiler IR that can handle inheritance from non-manifest base classes. In addition, the IR should satisfy a number of other important criteria. The IR should be expressive enough to support a wide range of statically typed surface languages from Java to Loom. The IR should be reasonably close to the machine and should be able to express efficient object representations (*e.g.*, shared method suites) and both static and dynamic method dispatch. The IR should enable optimizations based on simple and standard transformations. Lastly, the IR should be amenable to formal reasoning about compiler transformations and class linking.

G. Smolka (Ed.): ESOP/ETAPS 2000, LNCS 1782, pp. 135–149, 2000.

This paper presents $\lambda\mathit{ink}\varsigma$, which is an extension of the untyped λ-calculus that meets these design goals. $\lambda\mathit{ink}\varsigma$ extends the λ-calculus with *method suites,* which are ordered collections of methods; *slots,* which index into method suites; and *dictionaries,* which map method labels to slots. In $\lambda\mathit{ink}\varsigma$, method dispatch is implemented by first using a dictionary to find the method's slot and then using the slot to index into a method suite. $\lambda\mathit{ink}\varsigma$ can support true private names and avoid the fragile base class problem by dynamically changing the dictionary associated with an object when the object's view of itself changes [RS98,FR99b]. Separating dynamic dispatch into two pieces also enables more compiler optimizations. In this paper, we treat $\lambda\mathit{ink}\varsigma$ as a compiler IR, although the reader should think of it as more of a framework or basis for a compiler's IR.

By design, $\lambda\mathit{ink}\varsigma$ satisfies our goals. Because of the abstractions in $\lambda\mathit{ink}\varsigma$, it can express a wide range of surface class mechanisms, from the static classes found in Java through the dynamic inheritance of Loom (Section 5). By making $\lambda\mathit{ink}\varsigma$ untyped, we avoid limiting the applicability of $\lambda\mathit{ink}\varsigma$ to languages with incompatible type systems. The operations in the calculus allow compilers to leverage static information to optimize message dispatch. For example, the type system in C^{++} guarantees the slot at which each method may be found at run-time. In $\lambda\mathit{ink}\varsigma$, we may use this information to evaluate the dictionary lookup operation associated with message dispatch at compile time — providing the expected efficiency for message dispatch to C^{++} programmers. Because $\lambda\mathit{ink}\varsigma$ is based on the λ-calculus, familiar λ-calculus optimizations apply immediately to $\lambda\mathit{ink}\varsigma$ (Section 6), and these optimizations yield standard object-oriented optimizations when applied to $\lambda\mathit{ink}\varsigma$ programs. Consequently, ad-hoc optimizations for the object-oriented pieces of a compiler based on $\lambda\mathit{ink}\varsigma$ are not necessary. Because $\lambda\mathit{ink}\varsigma$ is a formal language, it is amenable to formal reasoning. For example, one can show that $\lambda\mathit{ink}\varsigma$ is confluent and that the reductions tagged as linking redexes are strongly normalizing (Section 4).

In the next section, we discuss the challenges involved in implementing inheritance from an unknown base-class. In Section 3, we present the syntax, operational semantics, and rewrite systems of $\lambda\mathit{ink}\varsigma$. To keep the discussion focused, we restrict the technical presentation to a version of $\lambda\mathit{ink}\varsigma$ with methods, but no fields (instance variables). The techniques used to handle methods apply directly to fields (see Section 5.1). Section 4 defines a simple class language Scl and shows how it can be translated to $\lambda\mathit{ink}\varsigma$. We prove that the translation of any “well-ordered" Scl program has the property that all linking steps can be reduced statically. In Section 5, we sketch how $\lambda\mathit{ink}\varsigma$ can serve as an IR for Moby, Loom, a mixin extension for Scl, and C^{++}. Section 6 further demonstrates the utility of the rewriting system for $\lambda\mathit{ink}\varsigma$ by showing how method dispatch can be optimized in the calculus. We conclude with a discussion of related and future work.

2 Inheritance from Unknown Classes

One of our principal design goals is to support inheritance from unknown base classes. Figure 1 shows where difficulties can arise when compiling languages with such a feature. The example is written in Moby, although similar examples can be written in Loom and OCaml. The module in Figure 1 defines a class `ColorPt` that extends an unknown base class `Pt.Point` by inheriting its `getX` and `getY` methods, overriding its `move` method, and adding a `color` field. When compiling the module, the compiler knows only that the `Pt.Point` superclass has three methods (`getX`, `getY`, and `move`). The compiler

```
signature PT {
  class Point : {
    public meth getX : Unit -> Int
    public meth getY : Unit -> Int
    public meth move : (Int, Int) -> Unit
  }
}

module ColorPtFn (Pt : PT) {
  class ColorPt {
    inherits Pt.Point
    field c : Color
    public meth move (x : Int, y : Int) -> Unit {
      if (self.c == Red)
        then super.move(2*x, 2*y)
        else super.move(x, y)
    }
  }
}
```

Fig. 1. Inheriting from an unknown superclass

does not know in what order these methods appear in the internal representation of the `Point` class, nor what other private methods and fields the `Point` class might have. As an example of inheritance from such a class, suppose we have a class `PolarPt` that implements the `Pt.Point` interface and has additional polar-coordinate methods `getTheta` and `getRadius`. When we apply the `ColorPtFn` module to `PolarPt`, we effectively hide the polar-coordinate methods, making them private and allowing their names to be reused for other, independent methods in `ColorPt` and its descendants. Such private methods, while hidden, are not forgotten, since they may be indirectly accessible from other visible methods (*e.g.*, the `PolarPt` class might implement `getX` in terms of polar coordinates). This hiding is a problem when compiling the `PolarPt` class, since its code must have access to methods that might not be directly available in its eventual subclasses.

3 λ*inkς*

λ*inkς* is a λ-calculus with method suites, slots, and dictionaries, which provides a notation for class assembly, inheritance, dynamic dispatch, and other object-oriented features.

3.1 Syntax

The syntax of λ*inkς* is given by the grammar in Figure 2. In addition to the standard λ-calculus forms, there are eight expression forms for supporting objects and classes. The term $\langle e_1, \ldots, e_n \rangle$ constructs a method suite from the expressions $e_1, \ldots, e_n$, where each e_i is assigned slot i. The expression $e@e'$ extracts the value stored in the slot denoted by

$$
\begin{array}{rlr}
e ::= & x & \text{variable} \\
\mid & \lambda x.e \quad \mid \quad e(e') & \text{function abstraction/application} \\
\mid & (e_1, ..., e_n) \quad \mid \quad \pi_i\, e & \text{tuple creation/projection} \\
\mid & \langle e_1, \ldots, e_n \rangle & \text{method suite construction} \\
\mid & e@e' & \text{method suite indexing} \\
\mid & e || e' & \text{method suite extension} \\
\mid & e@e' \leftarrow e'' & \text{method override} \\
\mid & i & \text{slot} \\
\mid & e + e' & \text{slot addition} \\
\mid & \{m_1 \mapsto e_1, \ldots, m_n \mapsto e_n\} & \text{dictionary construction} \\
\mid & e!m & \text{dictionary application}
\end{array}
$$

Fig. 2. The syntax of $\lambda\mathit{ink}_\varsigma$

$$
\begin{array}{rcl}
(\lambda x.e)(v) & \hookrightarrow & e[x \mapsto v] \\
\pi_i(v_1, \ldots, v_n) & \hookrightarrow & v_i \quad \textit{where } 1 \leq i \leq n \\
i + j & \hookrightarrow & k \quad \textit{where } k = i + j \\
\{m_1 \mapsto v_1, \ldots, m_n \mapsto v_n\}!m_i & \hookrightarrow & v_i \quad \textit{where } 1 \leq i \leq n \\
\langle v_1, \ldots, v_n \rangle \,||\, \langle v_1', \ldots, v_{n'}' \rangle & \hookrightarrow & \langle v_1, \ldots, v_n, v_1', \ldots, v_{n'}' \rangle \\
\langle v_1, \ldots, v_i, \ldots, v_n \rangle @ i \leftarrow v' & \hookrightarrow & \langle v_1, \ldots, v', \ldots, v_n \rangle \qquad \textit{where } 1 \leq i \leq n \\
\langle v_1, \ldots, v_n \rangle @ i & \hookrightarrow & v_i \quad \textit{where } 1 \leq i \leq n
\end{array}
$$

Fig. 3. Reduction rules for $\lambda\mathit{ink}_\varsigma$

e' from the method suite denoted by e. The method suite extension $e||e'$ concatenates the suites e and e'. The last method suite operation is override, which functionally updates a slot in a given suite to produce a new suite. A slot is specified by a slot expression, which is either an integer i or the addition of two slot expressions. The expression $\{m_1 \mapsto e_1, \ldots, m_n \mapsto e_n\}$ denotes a dictionary where each label m_i is mapped to the slot denoted by e_i. Application of a dictionary to a label m is written $e!m$.

We identify terms up to the renaming of bound variables and use $e[x \mapsto e']$ to denote the capture-free substitution of e' for x in e. We assume that dictionaries are unordered and must represent finite functions. For instance, the dictionary $\{m \mapsto 1, m \mapsto 2\}$ is an ill-formed expression, since it maps m to two different values. To simplify notation, we use the following shorthands:

$$
\begin{array}{lll}
\texttt{let } x = e \texttt{ in } e' & \textit{for} & (\lambda x.e')(e) \\
\lambda(x_1, \ldots, x_n).e & \textit{for} & \lambda p.((\lambda x_1. \cdots \lambda x_n.e)\,(\pi_1\, p) \cdots (\pi_n\, p))
\end{array}
$$

3.2 Operational Semantics

We specify the operational semantics of $\lambda\mathit{ink}_\varsigma$ using an evaluation-context based rewrite system [FF86]. Such systems rewrite terms step-by-step until no more steps can be taken.

At each step, the term to be reduced is parsed into an evaluation context and a redex. The redex is then replaced, and evaluation begins anew with another parsing of the term. Note that since λ*ink*$_\varsigma$ is untyped there are legal expressions, such as $\pi_i\ (\lambda x.e)$, that cannot be reduced.

Two grammars form the backbone of the semantics. The first describes values, a subset of expressions that are in reduced form:

$$v ::= x \mid \lambda x.e \ \mid\ (v_1, \ldots, v_n) \ \mid\ \langle v_1, \ldots, v_n \rangle \ \mid\ i \ \mid\ \{m_1 \mapsto v_1, \ldots, m_n \mapsto v_n\}$$

The second grammar describes the set of evaluation contexts.

$$\begin{array}{rl} E ::= & [\cdot] \ \mid\ E(e) \ \mid\ v(E) \ \mid\ \pi_i\ E \\ \mid & \langle v_1, \ldots, E, \ldots, e_n \rangle \ \mid\ E||e \ \mid\ v||E \\ \mid & E@e \leftarrow e \ \mid\ v@E \leftarrow e \ \mid\ v@v \leftarrow E \ \mid\ E@e \ \mid\ v@E \\ \mid & E + e \ \mid v + E \ \mid\ \{m_1 \mapsto v_1, \ldots, m_i \mapsto E, \ldots, m_n \mapsto e_n\} \ \mid\ E!m \end{array}$$

The primitive reduction rules for λ*ink*$_\varsigma$ are given in Figure 3. We write $e \longmapsto e'$ if $e = E[e_0]$, $e' = E[e_0']$, and $e_0 \hookrightarrow e_0'$ by one of the rules above.

3.3 Reduction System

Under the operational semantics, there is no notion of transforming a program before it is run: all reductions happen when they are needed. We want, however, a method for rewriting λ*ink*$_\varsigma$ terms to equivalent, optimized versions. The basis of the rewrite system is the relation $\hookrightarrow$. We write $\rightarrow$ for the congruence closure of this relation; *i.e.*, for the system in which rewrites may happen anywhere inside a term. For example, reductions like $(\lambda x.\pi_1\ (v_1, x))(e) \rightarrow (\lambda x.v_1)(e)$ are possible, whereas in the operational semantics they are not. We write $\rightarrow^*$ for the reflexive, transitive closure of $\rightarrow$.

The reduction system will be used in the next two sections when we discuss static linking for a simple class language and optimizations. The reduction relation $\rightarrow$ is non-deterministic: multiple paths may emanate from a single expression, but it is confluent.

Theorem 1 *If $e \rightarrow^* e'$ and $e \rightarrow^* e''$, there is an e''' such that $e' \rightarrow^* e'''$ and $e'' \rightarrow^* e'''$.*

The proof uses the Tait-Martin-Löf parallel moves method [Bar84]; we omit the proof.

4 A Simple Class Language

To give evidence of the expressivity of λ*ink*$_\varsigma$, we now give a translation of a simple class-based language into λ*ink*$_\varsigma$. Simpler translations may be possible, but the translation here illustrates some techniques that are useful for more complex languages.

The source language is called SCL for "simple class language." The syntax of SCL appears in Figure 4. A program consists of a sequence of one or more class declarations followed by an expression; class declarations may only use those declarations that appear before and may not be recursive. There are two forms of class declaration. The first is a *base-class declaration*, which defines a class as a collection of methods. The

$$\begin{array}{rcll}
\textit{prog} & ::= & \textit{dcl prog} & \text{Programs}\\
 & | & \textit{exp} & \\
\textit{dcl} & ::= & \texttt{class}\,C\,\texttt{\{}\ \textit{meths}\ \texttt{\}} & \text{Class declarations}\\
 & | & \texttt{class}\,C\,\texttt{\{ inherit}\,C'\,:\texttt{\{}\ m^*\ \texttt{\}}\ \textit{meths}\ \texttt{\}} & \\
\textit{meths} & ::= & \epsilon & \\
 & | & \textit{meth meths} & \\
\textit{meth} & ::= & m(x)\textit{exp} & \text{Methods}\\
\textit{exp} & ::= & x & \text{Expressions}\\
 & | & \texttt{self} & \\
 & | & \textit{exp} \Leftarrow m(\textit{exp}) & \\
 & | & \texttt{super} \Leftarrow m(\textit{exp}) & \\
 & | & \texttt{new}\,C &
\end{array}$$

Fig. 4. The syntax for SCL

second form is a *subclass declaration,* which defines a class by inheriting methods from a *superclass,* overriding some of them, and then adding new methods. The subclass constrains the set of methods it visibly inherits from its superclass by listing the names of such methods as { m^* }. Other method names can be called only by superclass, not subclass, methods. This operation—in essence, a restriction operation—resembles MOBY's support for private members [FR99b,FR99a] and subsumes mechanisms found in JAVA and other languages.

At the expression level, SCL contains only those features relevant to linking. Methods take exactly one argument and have expressions for bodies; expressions include `self`, method dispatch, super-method dispatch, and object creation. A more complete language would include other expression forms, *e.g.*, integers, booleans, and conditionals.

The translation from SCL into λ*ink*$_\varsigma$ fixes representations for classes, objects, and methods. Each fully-linked class is translated to a triple (σ, ϕ, μ), where σ is the size of the class (*i.e.*, the number of slots in its method suite), ϕ is a dictionary for mapping method names to method-suite indices, and μ is the class's method suite. Each object is translated to a pair of the object's method suite and a dictionary for resolving method names. Each method is translated into a *pre-method* [AC96]; *i.e.*, a function that takes `self` as its first parameter.

The translation is defined by the following functions:

$\mathcal{P}[\![\textit{prog}]\!]_\Gamma$	Program translation
$\mathcal{C}[\![\textit{dcl}]\!]_\Gamma$	Class translation
$\mathcal{M}[\![meth]\!]_{\mu_{super},\phi_{super},\phi_{self},\Gamma}$	Method translation
$\mathcal{E}[\![\textit{exp}]\!]_{\mu_{super},\phi_{super},\phi_{self},\Gamma}$	Expression translation

These functions take a *class environment* Γ as a parameter, which maps the name of a class to its λ*ink*$_\varsigma$ representation. A class environment is tuple of fully-linking classes. The symbol $\Gamma(C)$ denotes the position in the tuple associated with class C, and $\Gamma \pm \{C \mapsto e\}$ denotes the tuple with e bound to class C.

The translation of methods and expressions require more parameters than the translation of programs and classes. In addition to a class environment, the method and expression translation functions take additional parameters to translate `self` and `super`. In particular, the dictionary ϕ_{self} is used to resolve message sends to `self`, and the method suite μ_{super} and dictionary ϕ_{super} are used to translate `super` invocations. Each method is translated to a $\lambda \mathit{ink}_\varsigma$ pre-method as follows:

$$\mathcal{M}[\![m(x)\mathit{exp}]\!]_{\mu_{\mathit{super}},\phi_{\mathit{super}},\phi_{\mathit{self}},\Gamma} = \lambda(\mathit{self}, x).\mathcal{E}[\![\mathit{exp}]\!]_{\mu_{\mathit{super}},\phi_{\mathit{super}},\phi_{\mathit{self}},\Gamma}$$

Expressions are translated as follows:

$$\begin{aligned}
\mathcal{E}[\![x]\!]_{\mu_{\mathit{super}},\phi_{\mathit{super}},\phi_{\mathit{self}},\Gamma} &= x \\
\mathcal{E}[\![\mathtt{self}]\!]_{\mu_{\mathit{super}},\phi_{\mathit{super}},\phi_{\mathit{self}},\Gamma} &= (\pi_1 \mathit{self}, \phi_{\mathit{self}}) \\
\mathcal{E}[\![\mathit{exp}_1 \Leftarrow m(\mathit{exp}_2)]\!]_{\mu_{\mathit{super}},\phi_{\mathit{super}},\phi_{\mathit{self}},\Gamma} &= \mathtt{let}\ \mathit{obj} \ = \ \mathcal{E}[\![\mathit{exp}_1]\!]_{\mu_{\mathit{super}},\phi_{\mathit{super}},\phi_{\mathit{self}},\Gamma} \\
&\ \mathtt{let}\ \mathit{meth} \ = \ (\pi_1\ \mathit{obj})@((\pi_2\ \mathit{obj})!m) \\
&\ \mathtt{in}\ \mathit{meth}(\mathit{obj}, \mathcal{E}[\![\mathit{exp}_2]\!]_{\mu_{\mathit{super}},\phi_{\mathit{super}},\phi_{\mathit{self}},\Gamma}) \\
&\ \text{where } \mathit{obj} \text{ and } \mathit{meth} \text{ are fresh} \\
\mathcal{E}[\![\mathtt{super} \Leftarrow m(\mathit{exp})]\!]_{\mu_{\mathit{super}},\phi_{\mathit{super}},\phi_{\mathit{self}},\Gamma} &= (\mu_{\mathit{super}}@(\phi_{\mathit{super}}!m))(\mathit{self},\ \mathcal{E}[\![\mathit{exp}]\!]_{\mu_{\mathit{super}},\phi_{\mathit{super}},\phi_{\mathit{self}},\Gamma}) \\
\mathcal{E}[\![\mathtt{new}\, C]\!]_{\mu_{\mathit{super}},\phi_{\mathit{super}},\phi_{\mathit{self}},\Gamma} &= \mathtt{let}\ (\sigma, \phi, \mu) \ = \ \Gamma(C)\ \mathtt{in}\ (\mu, \phi)
\end{aligned}$$

To translate `self`, we extract the method suite of *self* and pair it with the current self dictionary, ϕ_{self}. Note that because of method hiding, ϕ_{self} may have more methods than $(\pi_2\ \mathit{self})$ [RS98,FR99b]. To translate message sends, we first translate the receiver object and bind its value to *obj*. We then extract from this receiver its method suite $(\pi_1\ \mathit{obj})$ and its dictionary $(\pi_2\ \mathit{obj})$. Using dictionary application, we find the slot associated with method *m*. Using that slot, we index into the method suite to extract the desired pre-method, which we then apply to *obj* and the translated argument. We resolve `super` invocations by selecting the appropriate code from the superclass method suite according to the slot indicated in the superclass dictionary. Notice that this translation implements the standard semantics of super-method dispatch; *i.e.,* future overrides do not affect the resolution of super-method dispatch. We translate the `super` keyword to the ordinary variable *self*. In the translation of `new`, we look up the class to instantiate in the class environment. In our simple language, the new object is a pair of the class's method suite and dictionary.

The translation for subclasses appears in Figure 5. In the translation, certain subterms are annotated by a superscript L; these subterms denote link-time operations that are reduced during class linking. In addition, we use the function Names(meths) to extract the names of the methods in *meths*.

A subclass *C* is translated to a function *f* that maps any fully-linked representation of its base class *B* to a fully-linked representation of *C*. The body of the linking function *f* has three phases: slot calculation, dictionary definition, and method suite construction. In the first phase, fresh slot numbers are assigned to new methods (σ_n), while overridden (σ_{ov}) and inherited methods (σ_{inh}) are assigned the slots they have in *B*. The size of the subclass method suite (σ_C) is calculated to be the size of *B*'s suite plus the number of new methods. In the dictionary definition phase, each visible method name is associated with its slot number. During method suite construction, the definitions of overridden methods are replaced in the method suite for *B*. The function then extends the resulting method suite with the newly defined methods to produce the method suite for *C*.

$$
\begin{array}{l}
\mathcal{C}[\![\texttt{class}\, C\, \texttt{\{}\ \texttt{inherit}\, B : \texttt{\{}\ m^*\ \texttt{\}}\ \mathit{meths}\ \texttt{\}}]\!]_\Gamma = \\
\quad \lambda(\sigma_B, \phi_B, \mu_B). \\
\qquad\qquad \texttt{let}^L\ \sigma_{n_1} = \sigma_B +^L 1 \ldots\ \texttt{let}^L \sigma_{n_k} = \sigma_B +^L k \\
\qquad\qquad \texttt{let}^L\ \sigma_{\mathit{ov}_1} = \phi_B !^L \mathit{ov}_1\ \ \ldots\ \texttt{let}^L\ \sigma_{\mathit{ov}_j} = \phi_B !^L \mathit{ov}_j \\
\qquad\qquad \texttt{let}^L\ \sigma_{\mathit{inh}_1} = \phi_B !^L \mathit{inh}_1\ \ \ldots\ \texttt{let}^L\ \sigma_{\mathit{inh}_i} = \phi_B !^L \mathit{inh}_i \\
\qquad\qquad \texttt{let}^L\ \sigma_C = \sigma_B +^L k \\
\qquad\qquad \texttt{let}^L\ \phi_C = \{\, n_1 \mapsto \sigma_{n_1}, \ldots, n_k \mapsto \sigma_{n_k}, \\
\qquad\qquad\qquad\qquad\quad \mathit{ov}_1 \mapsto \sigma_{\mathit{ov}_1}, \ldots, \mathit{ov}_j \mapsto \sigma_{\mathit{ov}_j}, \\
\qquad\qquad\qquad\qquad\quad \mathit{inh}_1 \mapsto \sigma_{\mathit{inh}_1}, \ldots, \mathit{inh}_i \mapsto \sigma_{\mathit{inh}_i} \} \\
\qquad\qquad \texttt{let}^L\ \mu_0 = \mu_B \\
\qquad\qquad \texttt{let}^L\ \mu_1 = \mu_0 @^L \sigma_{\mathit{ov}_1} \leftarrow \mathcal{M}[\![\mathit{meth}_{\mathit{ov}_1}]\!]_{\mu_B, \phi_B, \phi_C, \Gamma} \\
\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad \texttt{let}^L\ \mu_j = \mu_{j-1} @^L \sigma_{\mathit{ov}_j} \leftarrow \mathcal{M}[\![\mathit{meth}_{\mathit{ov}_j}]\!]_{\mu_B, \phi_B, \phi_C, \Gamma} \\
\qquad\qquad \texttt{let}^L\ \mu_C = \mu_j \,||^L\, \langle \mathcal{M}[\![\mathit{meth}_{n_1}]\!]_{\mu_B, \phi_B, \phi_C, \Gamma}, \ldots, \mathcal{M}[\![\mathit{meth}_{n_k}]\!]_{\mu_B, \phi_B, \phi_C, \Gamma} \rangle \\
\qquad\qquad \texttt{in}\ (\sigma_C,\, \phi_C,\, \mu_C)
\end{array}
$$

where

$\textsc{NewNames} = \{n_1, \ldots, n_k\} = \text{Names}(\mathit{meths}) \setminus \{\ m^*\ \}$
$\textsc{OvNames} = \{\mathit{ov}_1, \ldots, \mathit{ov}_j\} = \{\ m^*\ \} \cap \text{Names}(\mathit{meths})$
$\textsc{InhNames} = \{\mathit{inh}_1, \ldots, \mathit{inh}_i\} = \{\ m^*\ \} \setminus \textsc{OvNames}$
$\{\mathit{meth}_{n_1}, \ldots, \mathit{meth}_{n_k}\} = \{m(x)\mathit{exp} \mid m(x)\mathit{exp} \in \mathit{meths} \text{ and } m \in \textsc{NewNames}\}$
$\{\mathit{meth}_{\mathit{ov}_1}, \ldots, \mathit{meth}_{\mathit{ov}_j}\} = \{m(x)\mathit{exp} \mid m(x)\mathit{exp} \in \mathit{meths} \text{ and } m \in \textsc{OvNames}\}$

Fig. 5. Translating Scl classes to λ*ink*ς

For base-class declarations, the translation is similar, except that there are no inherited or overridden methods. Furthermore, we use a special class $(0, \{\,\}, \langle\,\rangle)$ for the base-class argument. We omit the details for space reasons. Finally, we translate programs as follows:

$$
\begin{array}{r}
\mathcal{P}[\![\mathit{dcl}\ \mathit{prog}]\!]_\Gamma = \mathcal{P}[\![\mathit{prog}]\!]_{\Gamma'} \quad \text{where } \Gamma' = \Gamma \pm \{C \mapsto \mathcal{C}[\![\mathit{dcl}]\!]_\Gamma(\Gamma(B))\} \\
\mathcal{P}[\![\mathit{exp}]\!]_\Gamma = \mathcal{E}[\![\mathit{exp}]\!]_{\langle\,\rangle, \{\,\}, \{\,\}, \Gamma} \qquad\qquad\qquad\qquad\qquad\qquad
\end{array}
$$

The B stands for the base class in the definition of *dcl*.

The language Scl enjoys the property that for a *well-ordered program* — one in which all classes have been defined, and every class is defined before it is used — all linking operations labeled *L* can be eliminated statically. More formally,

Theorem 2 *If prog is a well-ordered program and* $\mathcal{P}[\![\mathit{prog}]\!]_\Gamma = e$*, then there is a term* e' *such that* $e \rightarrow^* e'$ *and* e' *contains no linking operations labeled* L*.*

This theorem can probably be proven using a size argument, but we use a strong-normalization approach instead. The proof of strong normalization is a bit subtle because expressions in λ*ink*ς can loop. We use a simple type system to show that a *fragment* of λ*ink*ς is strongly normalizing. The proof of strong normalization relies upon Tait's method [GLT89]. One may show that the translation of a well-ordered program is well-typed in the system, and hence all linking reductions can be done statically. We omit the proof for space reasons.

5 Other Examples

We now sketch how $\lambda ink\varsigma$ can be used to compile class mechanisms found in various programming languages.

5.1 Moby Classes

We originally designed $\lambda ink\varsigma$ to support Moby's class mechanism in a compiler that we are writing. Section 4's Scl models many of the significant parts of Moby's class mechanism, including one of its most difficult features to compile, namely its treatment of private names. In particular, Moby relies on signature matching in its module mechanism to hide private methods and fields [FR99a] (we illustrated this feature with the example in Section 2). Because Moby signatures define opaque interfaces, the Moby compiler cannot rely on complete representation information for the superclass of any class it is compiling. Instead, it must use the *class interface* of the superclass (*e.g.*, the `Pt` class in the `PT` signature) when compiling the subclass. Scl models this situation by requiring each subclass to specify in the `inherits` clause which superclass methods are visible.

The main piece missing from Scl are fields (*a.k.a.* instance variables), which require a richer version of $\lambda ink\varsigma$. While fields require extending the representation of objects with per-object instance variables, the details of instance variable access are very similar to those of method dispatch. As with methods, fields require dictionaries to map labels to slots and slot assignment. Dictionary creation and application are the same as for methods. When we create an object using `new`, we use the size of the class's instance variables as the size of the object to create — object initialization is done imperatively.

5.2 OCaml Classes

Like Moby, OCaml is a language with both parameterized modules and classes [Ler98]. For the most part, translating OCaml classes to $\lambda ink\varsigma$ is similar to translating Moby classes. The one difference is that OCaml supports a simple form of *multiple inheritance*, whereas Moby only has single inheritance. A class in OCaml can inherit from several base classes, but there is no sharing between base classes — the methods of the base classes are just concatenated. The one subtlety that we must address is that when compiling a class definition, we cannot assume that access to its methods will be zero-based in its subclasses. To solve this problem, we λ-abstract over the initial slot index. Otherwise, translating OCaml classes to $\lambda ink\varsigma$ is essentially the same as for Moby classes.[1]

5.3 Loom Classes

In the language Loom [BFP97], the class construct is an expression form, and a deriving class may use an arbitrary expression to specify its base class. Thus, unlike the translation in Section 4, a translation of Loom to our calculus cannot have the phase distinction between class link-time and run-time. In a translated Loom program, computation of slots, dictionary construction, method overrides, and method suite extensions can all happen at run-time. The fact that we can use one representation to handle both static and dynamic classes demonstrates the flexibility of our approach.

[1] To the best of our knowledge, the implementation techniques used for classes in the OCaml system have not been formalized or described in print, so we are not able to compare approaches.

5.4 Mixins

Mixins are functions that map classes to classes [FKF98] and, unlike parameterized modules, mixins properly extend the class that they are applied to (recall that applying `ColorPtFn` to `PolarPt` hid the polar-coordinate interface). Supporting this kind of class extension in $\lambda\mathit{ink}_\varsigma$ requires a bit of programming. The trick is to include a *dictionary constructor function* as an argument to the translated mixin. For example, consider the following mixin, written in an extension of Scl syntax:

```
mixin Print (C <: {show}) {
  meth print () { stdOut ⇐ print(self ⇐ show()) }
}
```

This mixin adds a `print` method to any class `C` that has a `show` method already. The translation of this mixin to $\lambda\mathit{ink}_\varsigma$ is similar to that of subclasses given in Section 4:

```
λ(σ_C,φ_C,μ_C,mkDict).
  let σ_print = σ_C+1
  let φ_Print = mkDict(φ_C, σ_print)
  let pre_print = λ(self).
      let print = (π_1 stdOut)@((π_2 stdOut)!print)
      let show = (π_1 self)@(φ_Print!show)
      in print(stdOut, show(self))
  let μ_Print = μ_C || ⟨pre_print⟩
  in (σ_print, φ_Print, μ_Print)
```

The main difference is that we use the `mkDict` function, supplied at the linking site, to create the extended dictionary. An alternative to this approach is to add a dictionary extension operation to $\lambda\mathit{ink}_\varsigma$. For purposes of this example, we assume that the surface language does not permit method-name conflicts between the argument class and the mixin, but it is possible to support other policies, such as C^{++}-style qualified method names, to resolve conflicts.

5.5 C^{++} and Java Classes

For a language with a manifest class hierarchy, such as C^{++} or Java, the language's static type system provides substantial information about the representation of dictionaries and method suites. By exploiting this representation information, we can optimize away all of the dictionary-related overhead in such programs, which results in the efficiency of method dispatch that C^{++} and Java programmers expect. The disadvantage of this approach is that it introduces representation dependencies that lead to the so-called *fragile base class* problem, in which changing the private representation of a base class forces recompilation of its subclasses. We should note that we do not know how to handle C^{++}'s form of multiple inheritance in $\lambda\mathit{ink}_\varsigma$ because of the object layout issues related to sharing of virtual base classes [Str94].

6 Optimization

Many compilers for higher-order languages use some form of λ-calculus as their intermediate representation (IR). In this section, we show that the techniques commonly used in λ-calculus-based compilers can be used to optimize our encoding of method dispatch in $\lambda\mathit{ink}_\varsigma$. Because $\lambda\mathit{ink}_\varsigma$ allows reuse of standard optimizations, the optimizer is simpler and more likely to be correct. It is important to note that the optimizations described in this section also apply to objects with instance variables. Even though instance variables are mutable, the optimizations focus on the dictionary and method-suite operations, which are *pure*. Consequently, the compiler is free to move these operations, subject only to the constraints of their data dependencies.

To make the discussion concrete, we consider the $\lambda\mathit{ink}_\varsigma$ representation of SCL programs and their optimization. In general, method dispatch in SCL requires an expensive lookup operation to map a method's label to its method-suite slot. Often, however, it is possible to apply transformations to reduce or eliminate this cost. We assume that we are optimizing well-typed programs that do not have run-time type errors (see Fisher and Reppy [FR99b] for an appropriate type system). We also assume that we produce the IR from SCL as described in Section 4, with the further step of normalizing the terms into a *direct-style* representation [FSDF93,Tar96,OT98] (a *continuation-passing style* representation [App92] is also possible). In this IR, all intermediate results are bound to variables, and the right-hand side of all bindings involve a single function application or primitive operation applied to *atomic* arguments (*i.e.*, either variables or constants).

6.1 Applying CSE and Hoisting

Common subexpression elimination (CSE) is a standard optimization whereby two identical pure expressions are replaced by a single expression. When method invocations are expanded into the $\lambda\mathit{ink}_\varsigma$ representation, there are many opportunities for CSE optimizations. For example, if there are two method invocations to the same object, fetching its dictionary will be a common subexpression. If the method calls are to the same method, then the dictionary application and method suite indexing operations will be common subexpressions.

Another standard transformation is to hoist invariant expressions out of functions. When applied to method dispatch, this transformation amortizes the cost of a dictionary application over multiple function applications or loop iterations.[2]

6.2 Self-Method Dispatch

While CSE and hoisting apply to any method dispatch, we can do significantly better when we have a message sent to `self`. Recall that the translation of the self-method dispatch `self` $\Leftarrow m(\mathit{exp})$ into $\lambda\mathit{ink}_\varsigma$ is

```
let $obj$ = ($\pi_1$($self$), $\phi_{self}$)
let $meth$ = $\pi_1$($obj$) @ ($\pi_2$($obj$)!m)
in $meth$($obj$, $exp$)
```

Normalizing to our IR and applying the standard *contraction* phase [App92] gives the following:

```
let $\mu$ = $\pi_1$($self$)
let $obj$ = ($\mu$, $\phi_{self}$)
let $\sigma$ = $\phi_{self}$!m
let $meth$ = $\mu$@$\sigma$
in $meth$($obj$, $a$)
```

[2] Note that loops are represented as tail-recursive functions in this style of IR.

where a is the atom resulting from normalizing the argument expression. The expression ϕ_{self}!m is invariant in its containing premethod, and thus the binding of σ can be lifted out of the premethod. This transformation has the effect of moving the dictionary application from run-time to link-time and leaves the following residual:

let μ = π_1($self$)
let obj = (μ, ϕ_{self})
let $meth$ = μ@σ
in $meth$(obj, a)

While it is likely that a compiler will generate this reduced form directly from a source-program self-method dispatch, this optimization is useful in the case where other optimizations (*e.g.*, inlining) expose self-method dispatches that are not present in the source.

6.3 Super-Method Dispatch

Calls to superclass methods can be resolved statically, so there should be no run-time penalty for superclass method dispatch. While it is possible to “special-case" such method calls in a compiler, we can get the same effect by code hoisting. Recall that the translation of the super-method dispatch `super` $\Leftarrow m(exp)$ into λ*ink*ς is

(μ_{super} @ (ϕ_{super}!m)) ($self$, exp)

As before, we normalize to our IR and contract, which produces the following:

let σ = ϕ_{super}!m
let $meth$ = μ_{super}@σ
in $meth$($self$, a)

where a is the atom resulting from normalizing the argument expression. In this case, we can hoist both the dictionary application and the method-suite indexing out of the containing method, which leaves the term “$meth$($self$, a).” Thus, by using standard λ-calculus transformations, we can resolve super-method dispatch statically. Furthermore, if the superclass’s method suite is known at compile time, then the standard optimization of reducing a selection from a known record can be applied to turn the call into a direct function call. This reduction has the further effect of enabling the call to be inlined.

6.4 Using Static Analysis

The optimizations that we have described so far require only trivial analysis. More sophisticated analyses can yield better optimizations [DGC95]. For example, *receiver-class prediction* [GDGC95] may permit us to eliminate some dictionary applications in method dispatches (as we do already for self-method dispatch). There may also be source-language type information, such as `final` annotations, that can enable optimizations, such as static method resolution.

6.5 Final Code Generation

We intentionally left the implementation of dictionaries abstract in λ*ink*ς so that the optimization techniques described above can be used independently of their concrete representation. Depending on the properties of the source language, dictionaries might be tables [Rém92,DH95], a graph structure [CC98], or a simple list of method names. We might also use caching techniques to improve dispatch performance when there is locality [DS84]. We might also maintain information in the compiler as to the origin of the dictionary and use multiple representations, each tailored to a particular dictionary origin. For example, a JAVA compiler can distinguish between dictionaries that correspond to classes and dictionaries that correspond to interfaces. In the former case, the dictionary is known at class-load time and dictionary applications can be resolved when the class is loaded and linked. For interfaces, however, a dictionary might be implemented as an indirection table [LST99].

7 Related Work

There is other published research on IRs for compiling class-based languages. The Vortex project at the University of Washington, for instance, supports a number of class-based languages using a common optimizing back-end [DDG$^+$95]. The Vortex IR has fairly high-level operations to support classes: class construction and method dispatch are both monolithic primitives. λink_ς, on the other hand, breaks these operations into smaller primitives. By working at a finer level of granularity, λink_ς is able to support a wider range of class mechanisms in a single framework (*e.g.*, Vortex cannot support the dynamic classes found in LOOM).

Another approach pursued by researchers is to encode object-oriented features in typed λ-calculi. While such an approach can support any reasonable surface language design, its effectiveness as an implementation technique depends on the character of the encoding. For example, League, *et. al.*, have recently proposed a translation of a JAVA subset into the FLINT intermediate representation extended with row polymorphism [LST99]. Although they do not have an implementation yet, their encoding seems efficient, but it is heavily dependent on the semantics of JAVA. For example, their translation relies on knowing the exact set of interfaces that a class implements. The encoding approach has also been recently tried by Vanderwaart for LOOM [Van99]. In this case, because of the richness of LOOM's feature set, the encoding results in an inefficient implementation of operations like method dispatch. We believe that a compiler based on λink_ς can do at least as well for JAVA as the encoding approach, while doing much better for languages like MOBY and LOOM that do not have efficient encodings in the λ-calculus.

In other related work, Bono, *et. al.* have designed a class calculus, based on the λ-calculus, for evaluating single and mixin inheritance [BPS99]. The focus of their work differs from ours, in that their calculus describes the core functionality of a particular surface language, whereas we provide the basic building blocks with which to implement a myriad of surface designs. Essentially, their language could be implemented in λink_ς; the translation from their calculus to λink_ς would capture the implementation information encoded in their operational semantics.

There are other formal linking frameworks [Car97,Ram96,GM99,AZ99,DEW99]. Of particular relevance here are uses of β-reduction to implement linking of modules, as we do for the linking of classes. From the very beginning, the Standard ML of New Jersey compiler has used the λ-calculus to express module linking [AM87]. More recently, Flatt and Felleisen describe a calculus for separate compilation that maps *units* to functions over their free variables [FF98].

8 Conclusions

We have presented $\lambda\mathit{ink}\varsigma$, a low-level calculus for representing class-based object-oriented languages. $\lambda\mathit{ink}\varsigma$ satisfies the goals we set in designing an IR. In particular, it provides support for inheritance from non-manifest base classes, such as occurs in Moby, OCaml, and Loom. It is amenable to formal reasoning, such as in the proof of termination of linking in Section 4. As illustrated in Section 5, $\lambda\mathit{ink}\varsigma$ is expressive enough to support a wide-range of surface languages, from the concrete representations of Java to the dynamic classes of Loom. Finally, simple λ-calculus optimizations, such as common subexpression elimination and hoisting, yield standard object-oriented optimizations, such as method caching, when applied to $\lambda\mathit{ink}\varsigma$ terms.

We are currently implementing a compiler for Moby that uses $\lambda\mathit{ink}\varsigma$ as the basis of the object fragment of its IR. One refinement that we use in our implementation is to syntactically distinguish between the link-time and run-time forms of $\lambda\mathit{ink}\varsigma$. In the future, we plan to explore the use of $\lambda\mathit{ink}\varsigma$ to support dynamic class loading and mobile code, and to develop a typed IR based on $\lambda\mathit{ink}\varsigma$.

References

AC96. Abadi, M. and L. Cardelli. *A Theory of Objects*. Springer-Verlag, New York, NY, 1996.

AG98. Arnold, K. and J. Gosling. *The Java Programming Language*. Addison-Wesley, Reading, MA, 2nd edition, 1998.

AM87. Appel, A. W. and D. B. MacQueen. A Standard ML compiler. In *FPCA'87*, vol. 274 of *LNCS*, New York, NY, September 1987. Springer-Verlag, pp. 301–324.

App92. Appel, A. W. *Compiling with Continuations*. Cambridge University Press, Cambridge, England, 1992.

AZ99. Ancona, D. and E. Zucca. A primitive calculus for module systems. In *PPDP'99*, LNCS. Springer-Verlag, September 1999, pp. 62–79.

Bar84. Barendregt, H. P. *The Lambda Calculus*, vol. 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, revised edition, 1984.

BFP97. Bruce, K. B., A. Fiech, and L. Petersen. Subtyping is not a good "match" for object-oriented languages. In *ECOOP'97*, vol. 1241 of *LNCS*, New York, NY, 1997. Springer-Verlag, pp. 104–127.

BPS99. Bono, V., A. Patel, and V. Shmatikov. A core calculus of classes and mixins. In *ECOOP'99*, vol. 1628 of *LNCS*, New York, NY, June 1999. Springer-Verlag.

Car97. Cardelli, L. Program fragments, linking, and modularization. In *POPL'97*, January 1997, pp. 266–277.

CC98. Chambers, C. and W. Chen. Efficient predicate dispatching. *Technical report*, Department of Computer Science, University of Washington, 1998.

DDG^{+}95. Dean, J., G. DeFouw, D. Grove, V. Litvinov, and C. Chambers. Vortex: An optimizing compiler for object-oriented languages. In *OOPSLA'96*, October 1995, pp. 83–100.

DEW99. Drossopoulou, S., S. Eisenbach, and D. Wragg. A fragment calculus — towards a model of separate compilation, linking and binary compatibility. In *LICS-14*, June 1999, pp. 147–156.

DGC95. Dean, J., D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP'95*, August 1995.

DH95. Driesen, K. and U. Hölzle. Minimizing row displacement dispatch tables. In *OOPSLA'95*, October 1995, pp. 141–155.

DS84. Deutsch, L. P. and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *POPL'84*, January 1984, pp. 297–302.

FF86. Felleisen, M. and D. P. Friedman. Control operators, the SECD-machine, and the λ-calculus. In M. Wirsing (ed.), *Formal Description of Programming Concepts – III*, pp. 193–219. North-Holland, New York, N.Y., 1986.

FF98. Flatt, M. and M. Felleisen. Units: Cool modules for HOT languages. In *PLDI'98*, June 1998, pp. 236–248.

FKF98. Flatt, M., S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *POPL'98*, January 1998, pp. 171–183.

FR99a. Fisher, K. and J. Reppy. The design of a class mechanism for Moby. In *PLDI'99*, May 1999, pp. 37–49.

FR99b. Fisher, K. and J. Reppy. Foundations for MOBY classes. *Technical Memorandum*, Bell Labs, Lucent Technologies, Murray Hill, NJ, February 1999.

FSDF93. Flanagan, C., A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *PLDI'93*, June 1993, pp. 237–247.

GDGC95. Grove, D., J. Dean, C. Garrett, and C. Chambers. Profile-guided receiver class prediction. In *OOPSLA'95*, October 1995, pp. 108–123.

GLT89. Girard, J.-Y., Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, Cambridge, England, 1989.

GM99. Glew, N. and G. Morrisett. Type-safe linking and modular assembly language. In *POPL'99*, January 1999, pp. 250–261.

Ler98. Leroy, X. *The Objective Caml System (release 2.00)*, August 1998. Available from `http://pauillac.inria.fr/caml`.

LST99. League, C., Z. Shao, and V. Trifonov. Representing Java classes in a typed intermediate language. In *ICFP'99*, September 1999, pp. 183–196.

OT98. Oliva, D. P. and A. P. Tolmach. From ML to Ada: Strongly-typed language interoperability via source translation. *JFP*, **8**(4), July 1998, pp. 367–412.

Ram96. Ramsey, N. Relocating machine instructions by currying. In *PLDI'96*, May 1996, pp. 226–236.

Rém92. Rémy, D. Efficient representation of extensible records. In *ML'92 Workshop*, San Francisco, USA, June 1992. pp. 12–16.

RS98. Riecke, J. G. and C. Stone. Privacy via subsumption. In *FOOL5*, January 1998. A longer version will appear in *Information and Computation*.

RV98. Rémy, D. and J. Vouillon. Objective ML: An effective object-oriented extension to ML. *TAPOS*, **4**, 1998, pp. 27–50.

Str94. Stroustrup, B. *The Design and Evolution of C++*. Addison-Wesley, Reading, MA, 1994.

Str97. Stroustrup, B. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 3rd edition, 1997.

Tar96. Tarditi, D. *Design and implementation of code optimizations for a type-directed compiler for Standard ML*. Ph.D. dissertation, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, December 1996. Available as Technical Report CMU-CS-97-108.

Van99. Vanderwaart, J. C. Typed intermediate representations for compiling object-oriented languages, May 1999. Williams College Senior Honors Thesis.

Abstract Domains for Universal and Existential Properties

Andrew Heaton[1], Patricia M. Hill[2], and Andy King[3]

[1] School of Computer Studies, University of Leeds, LS2 9JT, UK,
heaton@scs.leeds.ac.uk,
Tel: +44 113 233 5322, Fax: +44 113 233 5468.
[2] School of Computer Studies, University of Leeds, LS2 9JT, UK,
hill@scs.leeds.ac.uk.
[3] Computing Laboratory, University of Kent at Canterbury, CT2 7NF, UK,
amk@ukc.ac.uk.

Abstract. Abstract interpretation theory has successfully been used for constructing algorithms to statically determine run-time properties of programs. Central is the notion of an abstract domain, describing certain properties of interest about the program. In logic programming, program analyses typically fall into two different categories: either they detect program points where the property definitely holds (universal analyses) or possibly holds (existential analyses). We study the relation between such analyses in the case where the concrete domain is a lattice join-generated by its set of join-irreducible elements. Although our intended application is for logic programming, the theory is sufficiently general for possible applications to other languages.

1 Introduction

Abstract interpretation theory has successfully been used for constructing algorithms to statically determine run-time properties of programs. Traditionally, the semantics of the program is specified with a concrete domain. The central notion is to approximate program semantics by defining an abstract domain whose operations mimic those of the concrete domain. The abstract domain describes certain properties of interest about the program. Each element of the abstract domain specifies information about a possibly infinite number of concrete states. Thus, in order to construct an abstract domain tracing a property of the program, the property needs to be considered as a property over sets of concrete states.

Our aim is to provide new techniques for the construction of new abstract domains from given ones. Many operations have been designed for systematically constructing new domains. Domain operators studied include reduced product [8,4], reduced power [8] and disjunctive completion [8,11]. Linear refinement is introduced in [13] as an extension of the Heyting completion studied in [14]. In [15], a new domain for freeness analysis of logic programs is defined using linear

G. Smolka (Ed.): ESOP/ETAPS 2000, LNCS 1782, pp. 150–164, 2000.

refinement. In this paper, we suppose that the concrete domain is a lattice join-generated by its set of join-irreducible elements. In this case, given any property p defined over each individual concrete state, p can always be uniformly extended to a property over sets of concrete states.

For example, in logic programming it is standard to define the concrete domain as the powerset of substitutions, $\wp(Sub)$, partially ordered by set inclusion. $\wp(Sub)$ is join-generated by Sub. For many properties of logic programs, it is natural to first define the property on substitutions and then lift the property to include sets of substitutions. Consider the property of groundness. A variable x is ground under a substitution $\theta \in Sub$ if θ binds x to a term with no variables. Letting X be the set of variables of interest, the mapping $gr : Sub \to \wp(X)$ is defined:

$$gr(\theta) = \{x \in X \mid var(\theta(x)) = \emptyset\}.$$

Suppose we now want to consider groundness as a property with domain $\wp(Sub)$. We can consider either definite (universal) groundness or possible (existential) groundness. For definite groundness, $Gr^{\forall} : \wp(Sub) \to \wp(X)$ is defined:

$$Gr^{\forall}(\Theta) = \bigcap\{gr(\theta) \mid \theta \in \Theta\}.$$

For possible groundness, $Gr^{\exists} : \wp(Sub) \to \wp(X)$ is defined:

$$Gr^{\exists}(\Theta) = \bigcup\{gr(\theta) \mid \theta \in \Theta\}.$$

Note that definite groundness traces positive information about the groundness of program variables, whereas possible groundness traces negative information. Knowledge of both positive and negative information about program properties such as groundness is particularly useful for debugging applications.

In general, given a concrete domain C, an abstract domain D and a property p mapping the join-irreducible elements of C to D, p is extended to C using the join operation of D. We name this extension of p the D-lattice property of p. For example, $Gr^{\forall}$ is the $D^{\forall}_{gr}$-lattice property of gr where $D^{\forall}_{gr}$ is the lattice $\wp(Sub)$, partially ordered by $\supseteq$ with set intersection as the join operation. $Gr^{\exists}$ is the $D^{\exists}_{gr}$-lattice property of gr where $D^{\exists}_{gr}$ is the lattice $\wp(Sub)$, partially ordered by $\subseteq$ with set union as the join operation.

The main theoretical results shown are as follows:

- Given a Galois connection (C, α, D, γ) (where C is completely distributive and join-generated by its set of join-irreducible elements) specifying an analysis tracing positive information of p, we show how to construct a mirror Galois connection $(C, \alpha^m, D^d, \gamma^m)$ (where D^d is the dual lattice of D) specifying an analysis tracing negative information of p.
- Suppose $op : C \to C$ is a concrete operation and $\langle D, op' \rangle$ is a correct abstract interpretation of $\langle C, op \rangle$ specified by (C, α, D, γ). We find conditions on $\langle D, op' \rangle$ and $\langle C, op \rangle$ which ensure that $\langle D^d, op' \rangle$ is a correct abstract interpretation of $\langle C, op \rangle$ specified by $(C, \alpha^m, D^d, \gamma^m)$.

The paper is organised as follows: in Section 3 we define the notion of lattice properties and mirror properties. Section 4 considers some applications with the well-known domains *Pos* and *Sharing* of logic programming. In Section 5 we consider the safe approximation of concrete functions in analyses for mirror properties. Finally, Section 6 gives some concluding remarks and directions for future work.

2 Preliminaries

Throughout the paper, we assume familiarity with the basic notions of lattice theory ([3]) and abstract interpretation ([7,8,9]). Below we introduce notation and recall some of the central notions.

2.1 Lattice Theory

In the following, we assume $\langle A, \sqsubseteq_A, \sqcap_A, \sqcup_A, \top_A, \bot_A \rangle$ is a complete lattice. The *dual* lattice $\langle A, \sqsubseteq^d_A, \sqcap^d_A, \sqcup^d_A, \top^d_A, \bot^d_A \rangle$ is defined such that:

1. $\forall a, b \in A.a \sqsubseteq^d_A b$ iff $b \sqsubseteq_A a$;
2. $\sqcap^d_A = \sqcup_A$;
3. $\sqcup^d_A = \sqcap_A$;
4. $\top^d_A = \bot_A$;
5. $\bot^d_A = \top_A$.

We will often write A^d to denote the dual lattice $\langle A, \sqsubseteq^d_A, \sqcap^d_A, \sqcup^d_A, \top^d_A, \bot^d_A \rangle$. Given a mapping $f : A_1 \to A_2$, we will sometimes abuse notation by also writing f to denote the dual mapping $f^d : A^d_1 \to A^d_2$ such that $f(a) = f^d(a)$ for all $a \in A_1$.

An element $a \in A$ is *join-irreducible* if, for any $S \subseteq A$, $a = \sqcup_A S$ implies $a \in S$. The set of join-irreducible elements of A is denoted by $JI(A)$. Letting $S \subseteq A$, then A is *join-generated* by S if, for all $a \in A$, $a = \sqcup_A \{x \in S \mid x \sqsubset_A a\}$. For convenience, we assume $\bot_A = \sqcup_A \emptyset$. An element $a \in A$ is an *atom* if a covers $\bot_A$, i.e. $a \neq \bot_A$ and $\forall x \in A.(\bot_A \sqsubset_A x \sqsubseteq_A a) \Rightarrow (x = a)$. We denote by $atom_A$ the set of atoms of A. Note that $atom_A \subseteq JI(A)$. A is *atomistic* if A is *join-generated* by $atom_A$. A is *dual-atomistic* if A^d is atomistic.

A complete lattice A is *completely distributive* if, for any $\{x_{i,k} \mid i \in I, k \in K(i)\} \subseteq A$, the following identity holds:

$$\bigsqcap_{i \in I} \bigsqcup_{k \in K(i)} x_{i,k} = \bigsqcup_{f \in I \rightsquigarrow K} \bigsqcap_{i \in I} x_{i,f(i)},$$

where for any $i \in I$, $K(i)$ is a set of indices, and $I \rightsquigarrow K$ is the set of all functions f from I to $\bigcup_{i \in I} K(i)$ such that $\forall i \in I.f(i) \in K(i)$.

Example 1. The powerset of any set S, $\wp(S)$, ordered with set-theoretic inclusion, is completely distributive and join-generated by S. In this case $\wp(S)$ is also an atomistic lattice where the atoms are the elements of S.

The key property of completely distributive lattices we shall use is:

Lemma 1 ([2]). Let A be a completely distributive lattice. Then, $x \in JI(A)$ iff for any $S \subseteq A$, $x \sqsubseteq_A \bigsqcup_A S$ implies $x \sqsubseteq_A s$ for some $s \in S$.

2.2 Galois Connections

If C and D are posets and $\alpha : C \rightarrow D$, $\gamma : D \rightarrow C$ functions such that $\forall c \in C. \forall d \in D. \alpha(c) \sqsubseteq_D d \Leftrightarrow c \sqsubseteq_C \gamma(d)$, then (C, α, D, γ) is a *Galois connection* between C and D. If in addition γ is 1-1, or, equivalently, α is onto then (C, α, D, γ) is a *Galois insertion* of D in C. In the setting of abstract interpretation, C and D are called the *concrete* and *abstract* domains, respectively. Given a Galois connection (C, α, D, γ), α and γ are uniquely determined by each other. A practical consequence of this is that an abstract interpretation can be performed by defining only one of α or γ. We assume that every concrete domain C and abstract domain D form complete lattices. Given a concrete domain C and an abstract domain D, a property is defined as a (partial) mapping from C to D. Every Galois connection (C, α, D, γ) can be viewed as a specification of the property $\alpha : C \rightarrow D$.

An important property of Galois connections is the preservation of bounds. Suppose C, D are complete lattices. A mapping $\alpha : C \rightarrow D$ is *additive* if it preserves least upper bounds. Thus if $S \subseteq C$ then $\alpha(\bigsqcup_C S) = \bigsqcup_D \{\alpha(c) \mid c \in S\}$. A mapping $\alpha : C \rightarrow D$ is *co-additive* if $\alpha : C^d \rightarrow D^d$ is additive. If (C, α, D, γ) is a Galois connection, then α is additive. The converse is also true, i.e. if α is additive then α entirely determines a unique Galois connection (C, α, D, γ). Thus in order to define a Galois connection (C, α, D, γ) (where C, D are complete lattices), it is sufficient to define an additive α.

One way of defining new Galois connections is by composition. Given two Galois connections $(C, \alpha_A, A, \gamma_A)$ and $(A, \alpha_D, D, \gamma_D)$, $(C, \alpha_A \circ \alpha_D, D, \gamma_D \circ \gamma_A)$ is a Galois connection. We call $(C, \alpha_A \circ \alpha_D, D, \gamma_D \circ \gamma_A)$ the *composition* of $(C, \alpha_A, A, \gamma_A)$ and $(A, \alpha_D, D, \gamma_D)$.

Suppose (C, α, D, γ) is a Galois connection and $op_C : C \rightarrow C$, $op_D : D \rightarrow D$ are operations on C and D, respectively. $\langle D, op_D \rangle$ is a *correct abstract interpretation* of $\langle C, op_C \rangle$ specified by (C, α, D, γ) if $\alpha(op_C(\gamma(d))) \sqsubseteq_D op_D(d)$ for all $d \in D$. $\langle D, op_D \rangle$ is *optimal* if $op_D = \alpha \circ op_C \circ \gamma$. If $\langle D, op_D \rangle$ is optimal, then op_D is the best approximation of op_C relative to D. $\langle D, op_D \rangle$ is *complete* if $\alpha \circ op_C = op_D \circ \alpha$. Completeness is a stronger property than optimality. Indeed, whenever $\langle D, op_D \rangle$ is complete, it can be shown that $op_D = \alpha \circ op_C \circ \gamma$ [10,12]. The completeness of op_C depends on D and is a property of the abstract domain.

If (C, α, D, γ) is a Galois insertion, each value of the abstract domain D is useful in the presentation of the concrete domain as all the elements of D represent distinct members of C. Moreover, any Galois connection may be lifted to a Galois insertion. This is done by identifying those values of the abstract domain with the same concrete meaning into an equivalence class. This process is known as *reduction* of the abstract domain. Each Galois insertion (C, α, D, γ) can equivalently be considered as an upper closure operator on C, $\rho = \gamma \circ \alpha$. For every Galois connection (C, α, D, γ), let $(C, \alpha_{\equiv}, D_{\equiv}, \gamma_{\equiv})$ be the Galois insertion obtained by reducing (C, α, D, γ). We associate the (upper) closure operator $\rho = \gamma_{\equiv} \circ \alpha_{\equiv}$ with (C, α, D, γ). The set of closure operators on C is partially ordered such that $\rho_1 \sqsubseteq \rho_2$ if $\forall c \in C.\ \rho_1(c) \sqsubseteq_C \rho_2(c)$. In this approach, the order relation on the set of closure operators on C corresponds to the order by means of which abstract domains are compared with regard to precision. More formally, if $(C, \alpha_1, D_1, \gamma_1)$ and $(C, \alpha_2, D_2, \gamma_2)$ are Galois connections with the associated closure operators ρ_1 and ρ_2, respectively, then we say D_1 is more precise than D_2 if $\rho_1 \sqsubseteq \rho_2$.

3 Properties of Programs

In abstract interpretation, Galois connections are used to specify properties of programs. To define a Galois connection (C, α, D, γ) between a concrete domain C and an abstract domain D, all we need to do is define an additive function $\alpha : C \to D$. It is well known that in the case where the concrete lattice C is join-generated by $JI(C)$, additive functions mapping C to an abstract domain D are completely determined by their values for join irreducible elements. More specifically, if $\alpha : C \to D$ is additive then

$$\alpha(c) = \bigsqcup_D \{\alpha(x) \mid x \in JI(C) \wedge x \sqsubseteq_C c\}.$$

Example 2. For logic programs, a standard choice of concrete lattice is the atomistic lattice $C_L = \langle \wp(Sub), \subseteq, \cap, \cup, \emptyset, Sub \rangle$, where Sub denotes the set of idempotent substitutions.

A program variable is ground if it is bound to a unique value. Groundness can be thought of as a property over Sub, i.e. as a property over $JI(C_L)$. Let X be the set of variables of interest. Then the set of variables ground under $\theta \in Sub$ is given by $gr : JI(C_L) \to \wp(X)$ defined

$$gr(\theta) = \{x \in X \mid var(\theta(x)) = \emptyset\}.$$

Let $\Theta \subseteq Sub$. The set of variables that are definitely ground under all $\theta \in \Theta$ is given by $Gr^{\forall} : C_L \to \wp(X)$ where

$$Gr^{\forall}(\Theta) = \{x \in X \mid \forall \theta \in \Theta. var(\theta(x)) = \emptyset\} = \bigcap \{gr(\theta) \mid \theta \in \Theta\}.$$

Alternatively, the set of variables that are possibly ground under all $\theta \in \Theta$ is given by $Gr^{\exists} : C_L \to \wp(X)$ where

$$Gr^{\exists}(\Theta) = \{x \in X \mid \exists \theta \in \Theta. var(\theta(x)) = \emptyset\} = \bigcup \{gr(\theta) \mid \theta \in \Theta\}. \qquad \square$$

Definition 1. Let C be a lattice. Then p is an *JI property* for C if there exists a set D such that p maps $JI(C)$ to D (denoted $p : JI(C) \to D$). □

Definition 2. Suppose C is join-generated by $JI(C)$ and let $p : JI(C) \to D$ be a JI property for C. Suppose D forms a complete lattice under the partial ordering $\sqsubseteq_D$. Then the *D-lattice* property of p, $P : C \to D$, is defined such that for every $c \in C$,

$$P(c) = \bigsqcup_D \{p(x) \mid x \in JI(C) \wedge x \sqsubseteq_C c\}.$$

Let D^d be the dual lattice of D. If P is the D-lattice property of p then we define the *mirror* property of P to be the D^d-lattice property of p. □

Note that the mirror of the mirror of P is P.

Example 3. Let D_{gr} be the complete lattice $(\wp(X), \subseteq, \cap, \cup, \emptyset, X)$. In Example 2, $Gr^{\exists}$ is the D_{gr}-lattice property of gr, and $Gr^{\forall}$ is the D^d_{gr}-lattice property of gr. Hence $Gr^{\forall}$ and $Gr^{\exists}$ are mirror properties. □

In the case where C is also a completely distributive lattice, we have the following theorem.

Theorem 1. Suppose C is a completely distributive lattice join generated by $JI(C)$ and D is a complete lattice. Let (C, α, D, γ) be a Galois connection. Then there exists α^m, γ^m such that

1. α^m is the mirror property of α.
2. $(C, \alpha^m, D^d, \gamma^m)$ is a Galois connection.

Proof. To prove 1, observe that as C is join-generated by $JI(C)$, for each $c \in C$,

$$\alpha(c) = \bigsqcup_D \{\alpha(x) \mid x \in JI(C) \wedge x \sqsubseteq_C c\}.$$

Hence by Definition 2,

$$\alpha^m(c) = \bigsqcap_D \{\alpha(x) \mid x \in JI(C) \wedge x \sqsubseteq_C c\}.$$

To prove 2, it is sufficient to show that α^m is additive. But

$$\begin{aligned} \alpha^m(\bigsqcup_C S) &= \bigsqcap_D \{\alpha(x) \mid x \in JI(C) \wedge x \sqsubseteq_C \bigsqcup_C S\} && \text{(by Definition 2)} \\ &= \bigsqcap_D \{\alpha(x) \mid x \in JI(C) \wedge x \sqsubseteq_C s \wedge s \in S\} && \text{(by Lemma 1)} \\ &= \bigsqcap_D \{\alpha^m(s) \mid s \in S\}. \end{aligned}$$

Hence α^m is additive. □

The compositional design of Galois connections is a method for specifying program properties by successive refinements. The following lemma gives a sufficient condition for the preservation of compositions of Galois connections between mirror properties.

Lemma 2. Suppose C is a completely distributive lattice join-generated by $JI(C)$, and A, D are complete lattices. Suppose $(C, \alpha_p, D, \gamma_p)$, $(C, \alpha_p^m, D^d, \gamma_p^m)$, $(C, \alpha_A, A, \gamma_A)$ and $(C, \alpha_A^m, A^d, \gamma_A^m)$ are Galois connections such that α_p, α_p^m and α_A, α_A^m are mirror properties. Also suppose $(A, \alpha_D, D, \gamma_D)$ is a Galois connection such that $(C, \alpha_p, D, \gamma_p)$ is the composition of $(C, \alpha_A, A, \gamma_A)$ and $(A, \alpha_D, D, \gamma_D)$. Then if α_D is co-additive, there exists $\gamma_D : D^d \to A^d$ such that $(A^d, \alpha_D, D^d, \gamma_D)$ forms a Galois connection and $(C, \alpha_p^m, D^d, \gamma_p^m)$ is the composition of $(C, \alpha_A^m, A^d, \gamma_A^m)$ and $(A^d, \alpha_D, D^d, \gamma_D)$.

Proof. First note that $\alpha_D : A \to D$ is co-additive implies that $\alpha_D : A^d \to D^d$ is additive, and so there exists $\gamma_D : D^d \to A^d$ such that $(A^d, \alpha_D, D^d, \gamma_D)$ forms a Galois connection.

To show that $(C, \alpha_p^m, D^d, \gamma_p^m)$ is the composition of $(C, \alpha_A^m, A^d, \gamma_A^m)$ and $(A^d, \alpha_D, D^d, \gamma_D)$, it is sufficient to show that $\alpha_p^m = \alpha_D \circ \alpha_A^m$. Suppose $c \in C$. By Definition 2,

$$\alpha_p^m(c) = \prod_D \{\alpha_p(x) \mid x \in JI(C) \wedge x \sqsubseteq_C c\}.$$

Now $\alpha_p(x) = \alpha_D(\alpha_A(x))$ and so

$$\alpha_p^m(c) = \prod_D \{\alpha_D(\alpha_A(x)) \mid x \in JI(C) \wedge x \sqsubseteq_C c\}.$$

But α_D is co-additive and so

$$\alpha_p^m(c) = \alpha_D(\prod_A \{\alpha_A(x) \mid x \in JI(C) \wedge x \sqsubseteq_C c\}) = \alpha_D(\alpha_A^m(c)).\square$$

Let ρ_p, ρ_A be the associated closure operators of $(C, \alpha_p, D, \gamma_p)$ and $(C, \alpha_A, A, \gamma_A)$, respectively. Note that whenever $(C, \alpha_p, D, \gamma_p)$ is the composition of $(C, \alpha_A, A, \gamma_A)$ and $(A, \alpha_D, D, \gamma_D)$, then $\rho_A \sqsubseteq \rho_p$. Thus Lemma 2 can be interpreted as giving a sufficient condition for the preservation of the relative precision between mirror properties, that is, when $\rho_A \sqsubseteq \rho_p$ implies $\rho_A^m \sqsubseteq \rho_p^m$ (where ρ_p^m, ρ_A^m are the associated closure operators of $(C, \alpha_A^m, A^d, \gamma_A^m)$ and $(C, \alpha_p^m, D^d, \gamma_p^m)$, respectively).

4 Applications

We consider the abstract domains *Pos* and *Sharing* from logic programming. In the following, let *Vars* denote a countable set of variables, and X denote a non-empty finite subset of *Vars* containing the variables of interest.

4.1 *Pos*

We briefly recall the definition of *Pos*. The domain *Pos* consists of the set of positive propositional formulae on X, where a propositional formula is positive if it is satisfied when every variable is assigned the value true. *Pos* is a lattice whose ordering is given by logical consequence, and the join and meet by logical disjunction and conjunction, respectively. Adding the bottom propositional formula $false$ to *Pos*, makes *Pos* a complete lattice. Letting C_L be the concrete domain defined in Example 2, the Galois insertion $(C_L, \alpha_{pos}, Pos, \gamma_{pos})$ is such that $\alpha_{pos} : C_L \to Pos$ where for all $\Theta \in C_L$,

$$\alpha_{pos}(\Theta) = \bigvee_{\theta \in \Theta} \bigwedge_{x \in X} \{x \leftrightarrow \bigwedge var(\theta(x))\}.$$

Note that α_{pos} is the *Pos*-lattice property of the *JI* property $p_{pos} : Sub \to Pos$ defined such that

$$p_{pos}(\theta) = \bigwedge_{x \in X} \{x \leftrightarrow \bigwedge var(\theta(x))\}.$$

The abstract unification function for *Pos*, $Unif^{pos} : Pos \times Pos \to Pos$, is given by logical conjunction, that is, the meet operation of *Pos*.

Recall that in Examples 2 and 3, definite groundness is specified by $Gr^{\forall}$. In fact $Gr^{\forall}$ maps C_L onto D^d_{gr} and so there exists $\gamma^{\forall}$ such that $(C_L, Gr^{\forall}, D^d_{gr}, \gamma^{\forall})$ forms a Galois insertion. This domain is originally due to Jones and Søndergaard [16]. In [18], when considering the concrete domain to be sets of substitutions closed by instantiation, it is shown that *Pos* can be constructed by using only the definition of groundness. More specifically, [18] shows that *Pos* is exactly the least abstract domain which contains all the (double) intuitionistic implications between elements of D^d_{gr}.

Let $\alpha_D : Pos \to D^d_{gr}$ be defined such that for all $\phi \in Pos$,

$$\alpha_D(\phi) = \{x \in X \mid \phi \models x\}.$$

Now α_D is additive since $\alpha_D(\phi_1 \vee \phi_2) = \alpha_D(\phi_1) \cap \alpha_D(\phi_2)$. Hence there exists γ_D such that $(Pos, \alpha_D, D^d_{gr}, \gamma_D)$ forms a Galois connection. Also $Gr^{\forall}(\Theta) = \alpha_D(\alpha_{pos}(\Theta))$ for all $\Theta \in C_L$, therefore $(C_L, Gr^{\forall}, D^d_{gr}, \gamma^{\forall})$ is the composition of $(C_L, \alpha_{pos}, Pos, \gamma_{pos})$ and $(Pos, \alpha_D, D^d_{gr}, \gamma_D)$.

The mirror property of $Gr^{\forall}$ is $Gr^{\exists}$. Now $Gr^{\exists}$ maps C_L onto D_{gr} and so there exists $\gamma^{\exists}$ such that $(C_L, Gr^{\exists}, D^d_{gr}, \gamma^{\exists})$ forms a Galois insertion.

The mirror property of α_{pos} is $\alpha^m_{pos} : C_L \to Pos^d$ where

$$\alpha^m_{pos}(\Theta) = \bigwedge_{\theta \in \Theta} \bigwedge_{x \in X} \{x \leftrightarrow \bigwedge var(\theta(x))\}.$$

Lemma 3. There exists γ^m_{pos} such that $(C_L, \alpha^m_{pos}, Pos^d, \gamma^m_{pos})$ forms a Galois connection. Also $(C_L, Gr^{\exists}, D_{gr}, \gamma^{\exists})$ is the composition of $(C_L, \alpha^m_{pos}, Pos^d, \gamma^m_{pos})$ and $(Pos^d, \alpha_D, D_{gr}, \gamma_D)$.

Proof. By Theorem 1 there exists γ^m_{pos} such that $(C_L, \alpha^m_{pos}, Pos^d, \gamma^m_{pos})$ forms a Galois connection. Now $\alpha_D(\phi \wedge \psi) = \alpha_D(\phi) \cup \alpha_D(\psi)$, and so $\alpha_D : Pos \to D^d_{gr}$ is co-additive. Therefore by Lemma 2, $(C_L, Gr^{\exists}, D_{gr}, \gamma^{\exists})$ is the composition of $(C_L, \alpha^m_{pos}, Pos^d, \gamma^m_{pos})$ and $(Pos^d, \alpha_D, D_{gr}, \gamma_D)$. □

Lemma 4. If $Card(X) \geq 2$, α^m_{pos} is not onto, thus $(C_L, \alpha^m_{pos}, Pos^d, \gamma^m_{pos})$ is not a Galois insertion.

Proof. By inspecting the definition of α^m_{pos}, it can be seen that $\alpha^m_{pos}(\Theta) \neq \bigvee X$ when $Card(X) \geq 2$, for any $\Theta \in C_L$. Hence α^m_{pos} is not onto. □

In order to obtain a Galois insertion, we apply the reduction process to Pos^d. $(C_L, \alpha^m_{pos}, Pos^d, \gamma^m_{pos})$ reduces to $(C_L, \alpha^m_{pos/\equiv}, Pos^d/\equiv, \gamma^m_{pos/\equiv})$ where for $\phi, \psi \in Pos^d$,

$$\phi \equiv \psi \Leftrightarrow \gamma^m_{pos}(\phi) = \gamma^m_{pos}(\psi), \quad \alpha^m_{pos/\equiv}(c) = \{\phi \mid \phi \equiv \alpha^m_{pos}(c)\}.$$

Let $\Gamma \subseteq Pos^d$ be defined such that

$$\Gamma = \{x \leftrightarrow \bigwedge\{y_1, \ldots, y_n\} \mid \forall 1 \leq i \leq n. x \neq y_i\}.$$

By inspecting the definition of α^m_{pos} (and noting that Sub is the set of idempotent substitutions, i.e. $\theta \in Sub$ implies $x \notin var(\theta(x))$ for all x), it can be seen that $Pos^d/\equiv$ is the lattice $\Lambda \subseteq Pos^d$ where Λ is the closure of Γ under conjunction. From Lemma 3 we obtain:

Theorem 2. $Pos^d/\equiv$ is more precise than D_{gr}.

Thus the precision ordering has been preserved for the mirror properties.

4.2 *Sharing*

We define *Sharing* as in [1]. We define the *set sharing* domain $SH = \wp(SG)$ where $SG = \{S \subseteq \wp(X) \mid \emptyset \notin S\}$. SH is partially ordered by set inclusion such that the join is given by set union and the meet by set intersection.

Let C_L be the concrete domain defined in Example 2. The set of variables occurring in a substitution θ through the variable v is given by the mapping $occs : Sub \times X \to \wp(X)$ defined such that

$$occs(\theta, x) = \{y \in X \mid x \in var(\theta(y))\}.$$

Given this, the Galois insertion $(C_L, \alpha_{sh}, SH, \gamma_{sh})$ specifying SH can be defined such that

$$\alpha_{sh}(\Theta) = \bigcup_{\theta \in \Theta} \{occs(\theta, x) \mid x \in Vars, occs(\theta, x) \neq \emptyset\}.$$

Note that α_{sh} is the SH-lattice property of the *JI* property $p_{sh} : Sub \to SH$ defined such that

$$p_{sh}(\theta) = \{occs(\theta, x) \mid x \in Vars, occs(\theta, x) \neq \emptyset\}.$$

For *Sharing*, the abstract unification function is defined as a mapping which captures the effects of a binding $x \to t$ on an element of SH. The definition uses the following three operations defined over SH.

The function $bin : SH \times SH \to SH$, called *binary union* is given by

$$bin(S_1, S_2) = \{s_1 \cup s_2 \mid s_1 \in S_1, s_2 \in S_2\}.$$

The *star-union* function $(\cdot)^* : SH \to SH$ is given by

$$S^* = \{s \in SG \mid \exists S' \subseteq S. s = \bigcup S'\}.$$

The *relevant component* function $rel : \wp(X) \times SH \to SH$ is given by

$$rel(V, S) = \{s \in S \mid s \cap V \neq \emptyset\}.$$

Let $v_x = \{x\}$, $v_t = var(t)$ and $v_{xt} = v_x \cup v_t$. Then

$$Unif^{sh}(S, x \to t) = (S \setminus (rel(v_{xt}, S)) \cup bin(rel(v_x, S)^*, rel(v_t, S)^*).$$

A domain for pair sharing is $PS = \wp(Pairs(X))$ where $Pairs(X) = \{\{x, y\} \mid x, y \in X,\ x \neq y\}$. PS is specified by the Galois insertion $(C_L, \alpha_{ps}, PS, \gamma_{ps})$, where

$$\alpha_{ps}(\Theta) = \bigcup_{\theta \in \Theta} \{\{x, y\} \in Pairs(X) \mid var(\theta(x)) \cap var(\theta(y)) \neq \emptyset\}.$$

Note that α_{ps} is the PS-lattice property of the *JI* property $p_{ps} : Sub \to PS$ defined such that

$$p_{ps}(\theta) = \{\{x, y\} \in Pairs(X) \mid var(\theta(x)) \cap var(\theta(y)) \neq \emptyset\}.$$

Defining $\alpha_{sp} : SH \to PS$ such that

$$\alpha_{sp}(S) = \bigcup \{Pairs(s) \mid s \in S\},$$

it follows that $\alpha_{ps}(\Theta) = \alpha_{sp}(\alpha_{sh}(\Theta))$ for all $\Theta \in C_L$. Also $\alpha_{sp}(S_1 \cup S_2) = \bigcup \{Pairs(s) \mid s \in S_1 \cup S_2\} = \alpha_{sp}(S_1) \cup \alpha_{sp}(S_2)$. Therefore α_{sp} is additive and so there exists γ_{sp} such that $(SH, \alpha_{sp}, PS, \gamma_{sp})$ forms a Galois connection. It follows that $(C_L, \alpha_{ps}, PS, \gamma_{ps})$ is the composition of $(C_L, \alpha_{sh}, SH, \gamma_{sh})$ and $(SH, \alpha_{sp}, PS, \gamma_{sp})$, and so PS is more abstract than SH.

The mirror property of α_{sh} is $\alpha^m_{sh} : C_L \to SH^d$ defined such that

$$\alpha^m_{sh}(\Theta) = \bigcap_{\theta \in \Theta} \{occs(\theta, x) \mid x \in Vars, occs(\theta, x) \neq \emptyset\}.$$

Lemma 5. There exists γ^m_{sh} such that $(C_L, \alpha^m_{sh}, SH^d, \gamma^m_{sh})$ forms a Galois insertion.

Proof. By Theorem 1, there exists γ^m_{sh} such that $(C_L, \alpha^m_{sh}, SH^d, \gamma^m_{sh})$ forms a Galois connection. To prove α^m_{sh} is onto, we show $\forall a \in SH^d. \exists \theta \in Sub. \alpha^m_{sh}(\{\theta\}) = a$ by induction on $Card(a)$.

The base case is when $a = \emptyset$. Let $\theta = \{x \to t \mid x \in X\}$ where t is a ground term. Then $\alpha^m_{sh}(\{\theta\}) = \emptyset$.

Suppose $\exists s \in a$ and let $a' = a \setminus \{s\}$. Using the induction hypothesis, $\exists \theta' \in Sub.\alpha^m_{sh}(\{\theta'\}) = a'$. Let $u \in Vars \setminus X$ be a variable such that $u \notin var(\theta'(x))$ for any $x \in X$. For every $y \in s$, suppose $\theta'(y) = t'_y$. Let t_y be a term such that $var(t_y) = var(t'_y) \cup \{u\}$. Then defining θ such that $\theta(x) = t_x$ for all $x \in s$ and $\theta(x) = \theta'(x)$ otherwise, $\alpha^m_{sh}(\{\theta\}) = a$. □

The mirror property of α_{ps} is $\alpha^m_{ps} : C_L \to PS^d$ defined such that

$$\alpha^m_{ps}(\Theta) = \bigcap_{\theta \in \Theta} \{\{x, y\} \in Pairs(X) \mid var(\theta(x)) \cap var(\theta(y)) \neq \emptyset\}.$$

Lemma 6. There exists γ^m_{ps} such that $(C_L, \alpha^m_{ps}, PS^d, \gamma^m_{ps})$ forms a Galois insertion.

Proof. By Theorem 1, there exists γ^m_{ps} such that $(C_L, \alpha^m_{ps}, PS^d, \gamma^m_{ps})$ forms a Galois connection. We show that α^m_{ps} is onto.

First suppose $a = \top_{ps} = Pairs(X)$. Let $u \in Vars \setminus X$. Then if $\theta(x) = u$ for every $x \in X$, $\alpha^m_{ps}(\{\theta\}) = Pairs(X)$ as required.

Suppose $a \neq \top_{ps}$. PS is dual-atomistic with $atom_{ps^d} = \{Pairs(X) \setminus \{\{x, y\}\} \mid \{x, y\} \in PS\}$. Therefore for every $a \neq \top_{ps}$, $a = \bigcap\{x \mid x \in atom_{ps^d} \wedge a \subseteq x\}$. But $\alpha^m_{ps}(\Theta) = \bigcap\{p_{ps}(\theta) \mid \theta \in \Theta\}$, and so it is sufficient to show that $\forall a \in atom_{ps^d}.\exists \theta \in Sub.p_{ps}(\theta) = a$.

Suppose $a = Pairs(X) \setminus \{\{x, y\}\}$ and let $u, v \in Vars \setminus X$. Defining θ such that $\theta(x) = u, \theta(y) = v$ and $\theta(z) = f(u, v)$ for every $z \in X \setminus \{x, y\}$, $p_{ps}(\theta) = a$. □

Theorem 3. If $Card(X) \geq 3$ then SH^m is not more precise than PS^m.

Proof. We need to show there exists $\Theta \in C_L$ such that $\gamma^m_{sh}(\alpha^m_{sh}(\Theta)) \not\subseteq \gamma^m_{ps}(\alpha^m_{ps}(\Theta))$.

Suppose $X = \{x, y, z\}$ (it is easy to generalise the proof for $Card(X) > 3$). Let $\Theta = \{\theta_1, \theta_2\}$ where $\theta_1 = \{x \to y, z \to y\}$ and $\theta_2 = \{x \to y\}$. It follows that $\gamma^m_{sh}(\alpha^m_{sh}(\{\theta_1, \theta_2\})) = \gamma^m_{sh}(\{\{x, y, x\}\} \cap \{\{x, y\}\}) = \gamma^m_{sh}(\emptyset) = Sub$. But $\gamma^m_{ps}(\alpha^m_{ps}(\{\theta_1, \theta_2\})) = \gamma^m_{ps}(\{\{x, y\}\}) \subset Sub$. Therefore $\gamma^m_{sh}(\alpha^m_{sh}(\Theta)) \not\subseteq \gamma^m_{ps}(\alpha^m_{ps}(\Theta))$. □

Thus in general the precision ordering is not preserved for mirror properties.

Theorem 4. PS^m is not more precise than SH^m.

Proof. We need to show there exists $\Theta \in C_L$ such that $\gamma^m_{ps}(\alpha^m_{ps}(\Theta)) \not\subseteq \gamma^m_{sh}(\alpha^m_{sh}(\Theta))$.

Let $\Theta = \{\epsilon\}$ where ϵ is the identity substitution. Now $\gamma^m_{sh}(\alpha^m_{sh}(\{\epsilon\})) = \gamma^m_{sh}(\{\{x\} \mid x \in X\}) \subset Sub$ and $\gamma^m_{ps}(\alpha^m_{ps}(\{\epsilon\})) = \gamma^m_{ps}(\emptyset) = Sub$. Therefore $\gamma^m_{ps}(\alpha^m_{ps}(\Theta')) \not\subseteq \gamma^m_{sh}(\alpha^m_{sh}(\Theta'))$. □

Hence the precision of SH^m and PS^m is not comparable in general.

5 Operations on Concrete Domains

When the concrete lattice C is join-generated by $JI(C)$, many operations on C can be defined in terms of operations on $JI(C)$.

Definition 3. Suppose C is join-generated by $JI(C)$. Then *op* is a JI operation if $op : JI(C) \times JI(C) \to JI(C)$[1]. For each concrete operation $Op : C \times C \to C$, we say *Op* is uniformly defined from a JI operation *op* if for all $c_1, c_2 \in C$,

$$Op(c_1, c_2) = \bigsqcup_C \{op(x_1, x_2) \mid x_1, x_2 \in JI(C) \wedge x_1 \sqsubseteq_C c_1 \wedge x_2 \sqsubseteq_C c_2\}.$$

Example 4. In logic programming, unification and projection can both be defined as JI operations $unif : Sub \times Sub \to Sub$, $proj_V : Sub \to Sub$ (for $V \subseteq Vars$) as follows:

$$unif(\theta_1, \theta_2) = mgu(eqn(\theta_1), eqn(\theta_2)),$$

$$proj_V(\theta) = \theta' \text{ where for each } x \in Vars,\ \theta'(x) = \begin{cases} \theta(x) & \text{if } x \in V \\ x & \text{otherwise} \end{cases}$$

where $eqn(\theta) = \{x = t \mid x \to t \in \theta\}$.

The concrete operations $Unif : C_L \times C_L \to C_L$ and $Proj_V : C_L \to C_L$ can be uniformly defined from *unif* and *proj* as follows:

$$Unif(\Theta_1, \Theta_2) = \bigcup \{unif(\theta_1, \theta_2) \mid \theta_1 \in \Theta_1 \wedge \theta_2 \in \Theta_2\},$$

$$Proj_V(\Theta) = \bigcup \{proj_V(\theta) \mid \theta \in \Theta\}. \qquad \square$$

Given an abstract operation Op_D, we show that if $\langle D, Op_D \rangle$ is a complete (and therefore also correct) abstract interpretation of $\langle C, Op \rangle$, then $\langle D, Op_D^d \rangle$ is a correct abstract interpretation of $\langle C, Op \rangle$.

Lemma 7. Suppose C, D are complete lattices and C is join-generated by $JI(C)$. Let $Op : C \times C \to C$ be a concrete operation uniformly defined from the JI operation $op : JI(C) \times JI(C) \to JI(C)$. Let $\langle D, Op_D \rangle$ be a complete abstract interpretation of *Op* specified by (C, α, D, γ). Then $\langle D^d, Op_D \rangle$ is a correct abstract interpretation of $\langle C, Op \rangle$ specified by $(C, \alpha^m, D^d, \gamma^m)$.

Proof. We need to show that $Op(\gamma^m(d_1), \gamma^m(d_2)) \sqsubseteq_C \gamma^m(Op_D(d_1, d_2))$ for all $d_1, d_2 \in D$.

Note that from Definition 3 it follows that *Op* is monotonic, i.e. if $c_1 \sqsubseteq_C c_1'$ and $c_2 \sqsubseteq_C c_2'$ then $Op(c_1, c_2) \sqsubseteq_C Op(c_1', c_2')$. Since $\langle D, Op_D \rangle$ is complete, $Op_D = \alpha \circ Op \circ \gamma$. Hence since Op, α, γ are all monotonic, Op_D is also monotonic. Now

[1] Note that to simplify the notation we assume that a JI operation has at most two input arguments. The results presented can easily be extended to operations with any number of arguments.

$Op(\gamma^m(d_1), \gamma^m(d_2)) =$
$\bigsqcup_C \{op(x_1, x_2) \mid x_1, x_2 \in JI(C) \wedge x_1 \sqsubseteq_C \gamma^m(d_1) \wedge x_2 \sqsubseteq_C \gamma^m(d_2)\}$.

Therefore it is sufficient to show that $op(x_1, x_2) \sqsubseteq_C \gamma^m(Op_D(d_1, d_2))$ for all $x_1, x_2 \in JI(C)$ such that $x_1 \sqsubseteq_C \gamma^m(d_1)$ and $x_2 \sqsubseteq_C \gamma^m(d_2)$. Now $x_1 \sqsubseteq_C \gamma^m(d_1)$ implies $\alpha^m(x_1) \sqsubseteq^d_D d_1$ and $x_2 \sqsubseteq_C \gamma^m(d_2)$ implies $\alpha^m(x_2) \sqsubseteq^d_D d_2$. Hence since Op_D is monotonic,

$$Op_D(\alpha^m(x_1), \alpha^m(x_2)) \sqsubseteq^d_D Op_D(d_1, d_2).$$

But $x_1, x_2 \in JI(C)$, thus $Op_D(\alpha^m(x_1), \alpha^m(x_2)) = Op_D(\alpha(x_1), \alpha(x_2))$. Since Op_D is complete,

$$Op_D(\alpha(x_1), \alpha(x_2)) = \alpha(Op(x_1, x_2)) = \alpha(op(x_1, x_2)).$$

By Definition 3, $op(x_1, x_2) \in JI(C)$ and so $\alpha(op(x_1, x_2)) = \alpha^m(op(x_1, x_2))$. Thus $\alpha^m(op(x_1, x_2)) \sqsubseteq^d_D Op_D(d_1, d_2)$ and so $op(x_1, x_2) \sqsubseteq_C \gamma^m(Op_D(d_1, d_2))$. □

Example 5. The abstract projection function for *Pos*, $Proj^{pos}_V : Pos \to Pos$, amounts to existentially quantifying a formula (see [6] for details). It is shown that $\langle Pos, Proj^{pos}_V \rangle$ is complete in Lemma 36 [6][2]. Therefore by Lemma 7, $\langle Pos^d, Proj^{pos}_V \rangle$ is a correct abstract interpretation of $\langle C_L, Proj_V \rangle$.

The abstract projection function for *Sharing*, $Proj^{sh}_V : SH \to SH$, is defined such that

$$Proj^{sh}_V(S) = \{s \cap V \mid s \in S\}$$

Theorem 5.2 [5] shows that $\langle SH, Proj^{sh}_V \rangle$ is complete. Therefore by Lemma 7, $\langle SH^d, Proj^{sh}_V \rangle$ is a correct abstract interpretation of $\langle C_L, Proj_V \rangle$.

On the other hand, [6] shows that $\langle Pos, Unif^{pos} \rangle$ is not complete and [5] shows that $\langle SH, Unif^{sh} \rangle$ is not complete. □

In fact, it can be shown that both $\langle Pos^d, Unif^{pos} \rangle$ and $\langle SH^d, Unif^{sh} \rangle$ are not correct abstract interpretations of $\langle C_L, Unif \rangle$.

Lemma 8. $\langle Pos^d, Unif^{pos} \rangle$ is not a correct abstract interpretation of $\langle C_L, Unif \rangle$.

Proof. It is sufficient to find $\phi \in Pos^d$ such that

$$Unif^{pos}(\phi, \phi) \not\models \alpha^m_{pos}(Unif(\gamma^m_{pos}(\phi), \gamma^m_{pos}(\phi))).$$

Let ϕ be the formula $x \leftrightarrow y$ and $\theta_1 = \{x \to f(1, y)\}$ and $\theta_2 = \{x \to f(y, 1)\}$. Note that $\theta_1, \theta_2 \in \gamma^m_{pos}(\phi)$. Now $unif(\theta_1, \theta_2) = \{x \to f(1, 1), y \to 1\}$ and so it follows that

$$\alpha^m_{pos}(Unif(\gamma^m_{pos}(\phi), \gamma^m_{pos}(\phi))) \models x \wedge y.$$

But $Unif^{pos}(\phi, \phi) = \phi$ and so $Unif^{pos}(\phi, \phi) \not\models \alpha^m_{pos}(Unif(\gamma^m_{pos}(\phi), \gamma^m_{pos}(\phi)))$, as required. □

[2] Note that in [6] and [5], *Pos* and *Sharing* are formulated differently from our presentation. In [6] and [5], however, it is evident that the proofs can be adapted.

Lemma 9. $\langle SH^d, Unif^{sh}\rangle$ is not a correct abstract interpretation of $\langle C_L, Unif\rangle$.

Proof. It is sufficient to find $S \in SH^d$ and a binding $x \to t$ such that

$$Unif^{sh}(S, x \to t) \nsubseteq \alpha^m_{sh}(Unif(\gamma^m_{sh}(S), \{\{x \to t\}\})).$$

Let $S = \{\{x, y\}\}$, $t = f(1, y)$ and $\theta = \{x \to f(y, 1)\}$. Note that $\theta \in \gamma^m_{sh}(S)$. Now $unif(\theta, \{x \to t\}) = \{x \to f(1, 1), y \to 1\}$ and so it follows that

$$\alpha^m_{sh}(Unif(\gamma^m_{sh}(S), \{\{x \to t\}\}) = \emptyset.$$

But $Unif^{sh}(S, x \to t) = \{\{x, y\}, \{x\}, \{y\}\}$ and so the result follows. □

Hence new abstract unification operations need to be devised for both Pos^d and SH^d.

6 Conclusion

We have shown how, given an abstract domain D specifying a lattice property α_p, an abstract domain D^d specifying the mirror property α^m_p can be constructed. We have also shown that if $\langle D, Op_D\rangle$ is a complete abstract interpretation of $\langle C, Op_C\rangle$, then $\langle D^d, Op_D\rangle$ is a correct abstract interpretation of $\langle C, Op_C\rangle$.

There are instances when non-complete abstract operations computing a property can be used to improve the precision of operations computing the mirror property. For example, formulae of the form $x \to y$ in *Pos* are interpreted as meaning "x ground implies y ground". The contrapositive of this is "y non-ground implies x non-ground". Thus this information could be used to improve the precision of a Pos^d analysis. In fact, since non-groundness information is approximated by freeness information, it would seem reasonable to implement Pos^d as a reduced product construction with *Pos* and a domain expressing freeness information. It would be interesting to see if generalisations of this method could be meaningfully applied to other domains. Another direction for future work is to see how our approach relates to lower/upper approximations used in concurrency [17].

Acknowledgments

We thank the anonymous referees for their useful comments. This work was supported by EPSRC Grant GR/M05645.

References

1. R. Bagnara, P. M. Hill, and E. Zaffanella. Set-sharing is redundant for pair-sharing. In P. Van Hentenryck, editor, *Static Analysis: Proceedings of the 4th International Symposium*, volume 1302, pages 53–67, Paris, France, 1997.

2. R. Balbes and P .Dwinger. *Distributive Lattices.* University of Missouri Press, Columbia, Missouri, 1974.
3. G. Birkhoff. *Lattice Theory.* AMS Colloquium Publication, Providence, RI, 3rd edition, 1967.
4. M. Codish, A. Mulkers, M. Bruynooghe, M. García de la Banda, and M. Hermenegildo. Improving abstract interpretations by combining domains. *ACM Transactions on Programming Languages and Systems*, 17(1):28–44, 1995.
5. A. Cortesi and G. Filè. Sharing is Optimal. *Journal of Logic Programming*, 38(3):371–386, 1999.
6. A. Cortesi, G. Filè, and W. Winsborough. Optimal Groundness Analysis Using Propositional Logic. *Journal of Logic Programming*, 27(2):137–167, 1996.
7. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. Fourth ACM Symp. Principles of Programming Languages*, pages 238–252, 1977.
8. P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proc. Sixth ACM Symp. Principles of Programming Languages*, pages 269–282, 1979.
9. P. Cousot and R. Cousot. Abstract Interpretation and Applications to Logic Programs. *Journal of Logic Programming*, 13(2 & 3):103–179, 1992.
10. R. Giacobazzi and F. Ranzato. Refining and Compressing Abstract Domains. In *Proceedings of the 24th International Colloquium on Automata, Languages and Programming ICALP 97*, volume 1256 of *Lecture Notes in Computer Science*, pages 771–781. Springer-Verlag, 1997.
11. R. Giacobazzi and .F Ranzato. Optimal Domains for Disjunctive Abstract Interpretation. *Science of Computer Programming*, 32:177–210, 1998.
12. R. Giacobazzi, F. Ranzato, and F. Scozzari. Making Abstract Interpretations Complete. *Journal of the ACM.* (to appear).
13. R. Giacobazzi, F. Ranzato, and F. Scozzari. Building Complete Abstract Interpretations in a Linear Logic-based Setting. In G. Levi, editor, *Static Analysis, Proceedings of the Fifth International Static Analysis Symposium SAS 98*, volume 1503 of *Lecture Notes in Computer Science*, pages 215–229. Springer-Verlag, 1998.
14. R. Giacobazzi and F. Scozzari. A Logical Model for Relational Abstract Domains. *ACM Transactions on Programming Languages and Systems*, 20(5):1067–1109, 1998.
15. P. Hill and F. Spoto. Freeness Analysis through Linear Refinement. In *Static Analysis: Proceedings of the 6th International Symposium*, volume 1694, pages 85–100, 1999.
16. N.D. Jones and H. Søndergaard. A Semantics-based Framework for the Abstract Interpretation of Prolog. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 123–142. Ellis Horwood Ltd, 1987.
17. F. Levi. A Symbolic Semantics for Abstract Model Checking. In *Static Analysis: Proceedings of the 5th International Symposium*, volume 1503, pages 134–151, 1998.
18. F. Scozzari. Logical Optimality of Groundness Analysis. In P. Van Hentenryck, editor, *Proceedings of International Static Analysis Symposium, SAS'97*, volume 1302 of *Lecture Notes in Computer Science*, pages 83–97. Springer-Verlag, 1997.

A Type System for Bounded Space and Functional In-Place Update—Extended Abstract

Martin Hofmann

LFCS Edinburgh, Mayfield Rd, Edinburgh EH9 3JZ, UK
mxh@dcs.ed.ac.uk

Abstract. We show how linear typing can be used to obtain functional programs which modify heap-allocated data structures in place.
We present this both as a "design pattern" for writing C-code in a functional style and as a compilation process from linearly typed first-order functional programs into malloc()-free C code.
The main technical result is the correctness of this compilation.
The crucial innovation over previous linear typing schemes consists of the introduction of a resource type $\diamond$ which controls the number of constructor symbols such as cons in recursive definitions and ensures linear space while restricting expressive power surprisingly little.
While the space efficiency brought about by the new typing scheme and the compilation into C can also be realised by with state-of-the-art optimising compilers for functional languages such as Ocaml [15], the present method provides guaranteed bounds on heap space which will be of use for applications such as languages for embedded systems or 'proof carrying code' [18].

1 Introduction

In-place modification of heap-allocated data structures such as lists, trees, queues in an imperative language such as C is notoriously cumbersome, error prone, and difficult to teach.

Suppose that a type of lists has been defined[1] in C by

```
typedef enum {NIL, CONS} kind_t;

typedef struct lnode {
  kind_t kind;
  int hd;
  struct lnode * tl;
} list_t;
```

and that a function

[1] Usually, one encodes the empty list as a NULL-pointer, whereas here it is encoded as a list_t with kind component equal to NIL. This is more in line with the encoding of trees we present below. If desired, we could go for the slightly more economical encoding, the only price being a loss of genericity.

G. Smolka (Ed.): ESOP/ETAPS 2000, LNCS 1782, pp. 165–179, 2000.

```
list_t reverse(list_t l)
```

should be written which reverses its argument "in place" and returns it. Everyone who has taught C will agree that even when recursion is used this is not an entirely trivial task. Similarly, consider a function

```
list_t insert(int a, list_t l)
```

which inserts a in the correct position in l (assuming that the latter is sorted) allocating one struct node.

Next, suppose, you want to write a function

```
list_t sort(list_t l)
```

which sorts its argument in place according to the insertion sort algorithm. Note that you cannot use the previously defined function insert() here as it allocates new space.

As a final example, assume that we have defined a type of trees

```
typedef struct tnode {
  kind_t kind;
  int label;
  struct tnode * left;
  struct tnode * right;
} tree_t;
```

(with kind_t extended with LEAF, NODE) and that we want to define a function

```
list_t breadth(tree_t t)
```

which constructs the list of labels of tree t in breadth-first order by consuming the space occupied by the tree and allocating at most one extra struct lnode. While again, there is no doubt that this can be done, my experience is that all of the above functions are cumbersome to write, difficult to verify, and likely to contain bugs.

Now compare this with the ease with which such functions are written in a functional language such as OCaml [15]. For instance,

```
let reverse l = let rec rev_aux l acc =
  match l with
      [] -> acc
  | a::l -> rev_aux l (a::acc)
in rev_aux l []

type tree = Leaf of int
          | Node of int*tree*tree

let rec breadth t = let rec breadth_aux l =
 match l with
  [] -> []
| Leaf(a)::t -> a::breadth_aux(t)
| Node(a,l,r)::t -> a::breadth_aux(t @ [l] @ [r])
in breadth_aux [t]
```

These definitions are written in a couple of minutes and are readily verified using induction and equational reasoning.

The difference, of course, is that the functional programs do not modify their argument in place but rather construct the result anew by allocating fresh heap space.

If the argument is not needed anymore it will eventually be reclaimed by garbage collection, but we have no guarantee whether and when this will happen. Accordingly, the space usage of a functional program will in general be bigger and less predictable than that of the corresponding C program.

The aim of this paper is to show that by imposing mild extra annotations one can have the best of both worlds: easy to write code which is amenable to equational reasoning, yet modifies its arguments in place and does not allocate heap space unless explicitly told to do so.

We will describe a linearly[2] typed functional programming language with lists, trees, and other heap-allocated data structure which admits a compilation into malloc()-free C. This may seem paradoxical at first sight because one should think that at least a few heap allocations would be necessary to generate initial data. However, our type system is such that while it does allow for the definition of *functions* such as the above examples, it does not allow one to define constant terms of heap-allocated type other than trivial ones like nil.

If we want to apply these functions to concrete data we either move outside the type system or we introduce an extension which allows for controlled introduction of heap space. However, in order to develop and verify functions as opposed to concrete computations doing so will largely be unnecessary.

This is made possible in a natural way through the presence of a special resource type $\diamond$ which in fact is the main innovation of the present system over earlier linear type systems, see Section 6.

While experiments with “hand-compiled” examples show that the generated C-code can compete with the highly optimised Ocamlopt native code compiler and outperforms the Ocaml run time system by far we believe that the efficient space usage can also be realised by state-of-the-art garbage collection and caching.

The main difference is that we can *prove* that the code generated by our compilation comes with an explicit bound on the heap space used (none at all in the pure system, a controllable amount in an extension with an explicit allocation operator). This will make our system useful in situations where space economy and guaranteed resource bounds are of the essence. Examples are programming languages for embedded systems (see [12] for a survey) or “proof-carrying code”.

In a nutshell the approach works as follows. The type $\diamond$ (dia_t in the C examples) gets translated into a pointer type, say void * whose values point to heap space of appropriate size to store one list or tree node. It is the task of the type system to maintain the invariant that overwriting such heap space does not affect the result.

[2] We always use “linear” in the sense of “affine linear”, i.e. arguments may be used at most once.

When invoking a recursive constructor function such as `cons()` or `node()` one must supply an appropriate number of arguments of type ◇ to provide the required heap space. Conversely, if in a recursion an argument of list or tree type is decomposed these ◇-values become available again.

Linear typing then ensures that overwriting the heap space pointed to by these ◇-values is safe.

It is important to realise that the `C` programs obtained as the target of the translation do not involve `malloc()` and therefore must necessarily update their heap allocated arguments in place. Traditional functional programs may achieve the same global space usage by clever garbage collection, but there will be no guarantee that under all circumstances this efficiency will be realised.

We also point out that while the language we present is experimental the examples we can treat are far from trivial: insertion sort, quick sort, breadth first traversal using queues, Huffman's algorithm, and many more. We therefore are lead to believe that with essentially engineering effort our system could be turned into a usable programming language for the abovementioned applications.

2 Functional Programming with C

Before presenting the language we show how the translated code will look like by way of some direct examples.

For the above-defined list type we would make the following definitions:

```
typedef void * dia_t;   and   list_t cons(dia_t d, int hd, list_t tl){
and                             list_t res;
list_t nil(){                   res.kind = CONS;
  list_t res;                   res.hd = hd;
  res.kind=NIL;                 *(list_t *)d = tl;
  return res;                   res.tl = (list_t *)d;
}                               return res;
                              }
```

followed by

```
typedef struct {   and   list_destr_t list_destr(list_t l) {
  kind_t kind;             list_destr_t res;
  dia_t d;                 res.kind = l.kind;
  int hd;                  if (res.kind == CONS) {
  list_t tl;                 res.hd = l.hd;
} list_destr_t;              res.d = (void *) l.tl;
                             res.tl = *l.tl;
                           }
                           return res;
                         }
```

The function `nil()` simply returns an empty list on the stack. The function `cons()` takes a pointer to free heap space (`d`), an entry (`hd`) and a list (`tl`) and returns on the stack a list with `hd`-field equal to `hd` and `tl`-field pointing to a heap location containing `tl`. This latter heap location is of course the one explicitly provided through the argument `d`.

The destructor function `list_destr()` finally, takes a list (`l`) and returns a structure containing a field `kind` with value `CONS` iff `l.kind` equals `CONS` and in this case containing in the remaining fields head and tail of `l`, as well as a pointer to a free heap location capable of storing a list node (`d`).

Once we have made these definitions we can implement `reverse()` in a functional style as follows:

```
list_t rev_aux(list_t l0, list_t acc) {
  list_destr_t l = list_destr(l0);
  return  l.kind==NIL ? acc
   : rev_aux(l.tl, cons(l.d, l.hd, acc));
}

list_t reverse(list_t l) {
  return rev_aux(l,nil());
}
```

Notice that `reverse()` updates its argument in place, as no call to `malloc()` is being made.

To implement `insert()` we need an extra argument of type `dia_t` since this function, just like `cons()`, increases the length. So we write:

```
list_t insert(dia_t d, int a, list_t l0) {
  list_destr_t l = list_destr(l0);
  return l.kind==NIL ? cons(d,a,nil())
   : a <= l.hd ? cons(d,a,cons(l.d,l.hd,l.tl))
     : cons(d,l.hd,insert(l.d,a,l.tl));
}
```

Using `insert()` we can implement insertion sort with in place modification as follows:

```
list_t sort(list_t l0) {
  list_destr_t l = list_destr(l0);
  return l.kind==NIL ? nil()
   : insert(l.d,l.hd,sort(l.tl));
}
```

Notice, how the value `l.d` which becomes available in decomposing `l` is used to feed the `insert()` function.

Finally, let us look at binary `int`-labelled trees. We define

```
tree_t leaf(int label) {
  tree_t res;
  res.kind = LEAF;
  res.label = label;
  return res;
}
```

and

```
tree_t node(dia_t d1, dia_t d2,
    int label, tree_t l, tree_t r) {
  tree_t res;
  res.kind = NODE;
  res.label = label;
  *(tree_t *)d1 = left;
  *(tree_t *)d2 = right;
  res.left = (tree_t *)d1;
  res.right = (tree_t *)d2;
  return res;
}
```

followed by

```
typedef struct {
  kind_t kind;
  int label;
  dia_t d1, d2;
  tree_t left, right;
} tree_destr_t;
```

and

```
tree_destr_t tree_destr(tree_t t) {
  tree_destr_t res;
  res.label = t.label;
  if((res.kind = t.kind) == NODE) {
    res.d1 = (dia_t)t.left;
    res.d2 = (dia_t)t.right;
    res.left = *(tree_t *)t.left;
    res.right = *(tree_t *)t.right;
  }
  return res;
}
```

Notice that we must pay *two* $\Diamond$s in order to build a tree node. In exchange, two $\Diamond$s become available when we decompose a tree.

To implement `breadth` we have to define a type `listtree_t` of lists of trees analogous to `list_t` with `int` replaced by `tree_t`. Of course, the associated helper functions need to get distinct names such as `niltree()`, etc.

We can then define a function `br_aux` with prototype

```
list_t br_aux(listtree_t l)
```

by essentially mimicking the functional definition above (the complete code is omitted for lack of space) and obtain the desired function `breadth` as

```
list_t breadth(dia_t d, tree_t t) {
  return br_aux(cons(d,t,nil()));
}
```

Notice that the type of `breadth` shows that the result requires one memory region more than the input.

All these functions do not use dynamic memory allocation because the heap space needed to store the result can be taken from the argument. To construct concrete lists in the first place we need of course dynamic memory allocation. The full paper shows how this can be accommodated in a controlled fashion. Of course, for these programs to be correct it is crucial that we do not overwrite heap space which is still in use. The main message of this paper is that this can be guaranteed systematically by adhering to a linear typing discipline.

In other words, a function must use its argument at most once.

For instance, the following code which attempts to double the size of its argument would be incorrect:

```
list_t twice(list_t l0) {
  list_destr_t l = list_destr(l0);

  return l.kind==NIL ? nil()
   : cons(l.d,0,(cons(l.d,0,twice(l.tl))));
}
```

Rather than returning a list of 0's twice the size of its input it returns a circular list! A similar effect happens, if we replace the last line of the code for `insert()` by

```
     cons(d,l.hd,insert(d,a,l.tl));
```

In each case the reason is the double usage of the $\Diamond$-values `d` and `l.d`.

3 A Linear Functional Programming Language

We will now introduce a linearly typed functional metalanguage and translate it systematically into `C`. This will be done with the following aims. First, it allows us to formally prove the correctness of the methodology sketched above, second it will relieve us from having to rewrite similar code many times. Suppose, for instance, you wanted to use lists of trees (as needed to implement breadth first search). Then all the basic list code (`list_t, nil(), cons()`, etc.) will have to be rewritten (this problem could presumably also be overcome through the use of `C`++ templates [13]). Thirdly, a formalised language with linear type system will allow us to enforce the usage restrictions on which the correctness of the above code relies. Finally, this will open up the possibility to extend the metalanguage to a fully-fledged functional language which would be partly compiled into `C` whenever this is possible and executed in the traditional functional way when this is not the case.

3.1 Syntax and Typing Rules

The zero-order types are given by the following grammar.

$$A ::= \mathsf{N} \mid \diamond \mid \mathsf{L}(A) \mid \mathsf{T}(A) \mid A_1 \otimes A_2$$

More type formers such as sum types, records, and variants can easily be added.

A first-order type is an expression of the form $T = (A_1, \ldots, A_n) \to B$ where $A_1 \ldots A_n$ and B are zero-order types.

A signature Σ is a partial function from identifiers (thought of as *function symbols*) to first-order types.

A *typing context* Γ is a finite function from identifiers (thought of as parameters) to zero order types; if $x \notin \mathrm{dom}(\Gamma)$ then we write $\Gamma, x{:}A$ for the extension of Γ with $x \mapsto A$. More generally, if $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Delta) = \emptyset$ then we write Γ, Δ for the disjoint union of Γ and Δ. If such notation appears in the premise of a rule below it is implicitly understood that these disjointness conditions are met.

Types not including $\mathsf{L}(-), \mathsf{T}(-), \diamond$ are called *heap-free*, e.g. N and $\mathsf{N} \otimes \mathsf{N}$ are heap-free.

Let Σ be a signature. The *typing judgement* $\Gamma \vdash_\Sigma e : A$ read "expression e has type A in typing context Γ and signature Σ" is defined by the following rules.

$$\frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash_\Sigma x : \Gamma(x)} \qquad (\textsc{Var})$$

$$\frac{\Sigma(f) = (A_1, \ldots, A_n) \to B \qquad \Gamma_i \vdash_\Sigma e_i : A_i \text{ for } i = 1 \ldots n}{\Gamma_1, \ldots, \Gamma_n \vdash_\Sigma f(e_1, \ldots, e_n) : B} \qquad (\textsc{Sig})$$

$$\frac{\Gamma, x{:}A, y{:}A \vdash_\Sigma e : B \qquad A \text{ heap-free}}{\Gamma, x{:}A \vdash_\Sigma e[x/y] : B} \qquad (\textsc{Contr})$$

$$\frac{c \text{ a } \mathtt{C} \text{ integer constant}}{\Gamma \vdash_\Sigma c : \mathsf{N}} \qquad (\textsc{Const})$$

$$\frac{\Gamma \vdash_\Sigma e_1 : \mathsf{N} \qquad \Delta \vdash_\Sigma e_2 : \mathsf{N} \quad \star \text{ a } \mathtt{C} \text{ infix opn.}}{\Gamma, \Delta \vdash e_1 \star e_2 : \mathsf{N}} \qquad (\textsc{Infix})$$

$$\frac{\Gamma \vdash_\Sigma e : \mathsf{N} \qquad \Delta \vdash_\Sigma e' : A \qquad \Delta \vdash_\Sigma e'' : A}{\Gamma, \Delta \vdash_\Sigma \mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ e'' : A} \qquad (\textsc{If})$$

$$\frac{\Gamma \vdash_\Sigma e : A \qquad \Delta \vdash_\Sigma e' : B}{\Gamma, \Delta \vdash_\Sigma e \otimes e' : A \otimes B} \qquad (\textsc{Pair})$$

$$\frac{\Gamma \vdash_\Sigma e : A \otimes B \qquad \Delta, x{:}A, y{:}B \vdash_\Sigma e' : C}{\Gamma, \Delta \vdash_\Sigma \mathsf{match}\ e\ \mathsf{with}\ x \otimes y {\Rightarrow} e' : C} \qquad (\textsc{Split})$$

$$\Gamma \vdash_\Sigma \mathsf{nil}_A : \mathsf{L}(A) \qquad (\textsc{Nil})$$

$$\frac{\Gamma_d \vdash_\Sigma e_d : \Diamond \qquad \Gamma_h \vdash_\Sigma e_h : A \qquad \Gamma_t \vdash_\Sigma e_t : \mathsf{L}(A)}{\Gamma_d, \Gamma_h, \Gamma_t \vdash_\Sigma \mathsf{cons}(e_d, e_h, e_t) : \mathsf{L}(A)} \qquad (\textsc{Cons})$$

$$\frac{\begin{array}{l}\Gamma \vdash_\Sigma e : \mathsf{L}(A) \\ \Delta \vdash_\Sigma e_{\mathsf{nil}} : B \\ \Delta, d{:}\Diamond, h{:}A, t{:}\mathsf{L}(A) \vdash_\Sigma e_{\mathsf{cons}} : B\end{array}}{\Gamma, \Delta \vdash_\Sigma \mathsf{match}\ e\ \mathsf{with}\ \mathsf{nil} {\Rightarrow} e_{\mathsf{nil}} | \mathsf{cons}(d, h, t) {\Rightarrow} e_{\mathsf{cons}} : B} \qquad (\textsc{List-Elim})$$

$$\frac{\Gamma \vdash_\Sigma e : A}{\Gamma \vdash_\Sigma \mathsf{leaf}(e) : \mathsf{T}(A)} \qquad (\textsc{Leaf})$$

$$\frac{\begin{array}{c}\Gamma_{d1} \vdash_\Sigma e_{d1} : \Diamond \qquad \Gamma_{d2} \vdash_\Sigma e_{d2} : \Diamond \qquad \Gamma_a \vdash_\Sigma e_a : A \\ \Gamma_l \vdash_\Sigma e_l : \mathsf{T}(A) \qquad \Gamma_r \vdash_\Sigma e_r : \mathsf{T}(A)\end{array}}{\Gamma_{d1}, \Gamma_{d2}, \Gamma_a, \Gamma_l, \Gamma_r \vdash_\Sigma \mathsf{node}(e_{d1}, e_{d2}, e_a, e_l, e_r) : \mathsf{T}(A)} \qquad (\textsc{Node})$$

$$\frac{\begin{array}{c}\Gamma \vdash_\Sigma e : \mathsf{T}(A) \qquad \Delta, a{:}A \vdash_\Sigma e_{\mathsf{leaf}} : B \\ \Delta, d_1{:}\Diamond, d_2{:}\Diamond, a{:}A, l{:}\mathsf{T}(A), r{:}\mathsf{T}(A) \vdash_\Sigma e_{\mathsf{node}} : B\end{array}}{\Gamma, \Delta \vdash_\Sigma \mathsf{match}\ e\ \mathsf{with}\ \mathsf{leaf}(a) {\Rightarrow} e_{\mathsf{leaf}} | \mathsf{node}(d_1, d_2, a, l, r) {\Rightarrow} e_{\mathsf{node}} : B} \qquad (\textsc{Tree-Elim})$$

Remarks The symbol $\star$ in rule Infix ranges over a set of binary infix operations such as +, - ,/ , *, <=, ==, . . . We may include more such operations and also other base types such as floating point numbers or characters.

As usual, we omit type annotations wherever possible. The constructs involving match bind variables.

Application of function symbols or operations to their operands is linear in the sense that several operands must in general not share common free variables. This is because of the implicit side condition on juxtaposition of contexts mentioned above. In view of rule Contr, however, variables of a heap-free type may be shared and moreover the same free variable may appear in different branches of a case distinction as follows e.g. from the form of rule If. Here is how we typecheck $x + x$ when $x{:}\mathsf{N}$. First, we have $x{:}\mathsf{N} \vdash x : \mathsf{N}$ and $y{:}\mathsf{N} \vdash y : \mathsf{N}$ by Var. Then $x{:}\mathsf{N}, y{:}\mathsf{N} \vdash x{+}y : \mathsf{N}$ by Infix and finally $x{:}\mathsf{N} \vdash x{+}x : \mathsf{N}$ by rule Contr. It follows by standard type-theoretic techniques that typechecking for this system is decidable in linear time.

Programs A *program* consists of a signature Σ and for each symbol

$$f : (A_1, \ldots, A_n) {\to} B$$

contained in Σ a term

$$x_1{:}A_1, \ldots, x_n{:}A_n \vdash_\Sigma e_f : B$$

3.2 Set-Theoretic Interpretation

In order to specify the purely functional meaning of programs we introduce a set-theoretic interpretation as follows: types are interpreted as sets by

$$\begin{aligned}
&[\![\mathsf{N}]\!] = \mathbf{Z}\\
&[\![\Diamond]\!] = \{0\}\\
&[\![\mathsf{L}(A)]\!] = \text{finite lists over } [\![A]\!]\\
&[\![\mathsf{T}(A)]\!] = \text{binary } [\![A]\!]\text{-labelled trees}\\
&[\![A \otimes B]\!] = [\![A]\!] \times [\![B]\!]
\end{aligned}$$

To each program $(\Sigma, (e_f)_{f \in \mathrm{dom}(\Sigma)})$ we can now associate a mapping ρ such that $\rho(f)$ is a *partial* function from $[\![A_1]\!] \times \ldots [\![A_n]\!]$ to $[\![B]\!]$ for each $f : (A_1, \ldots, A_n) \to B$.

This meaning is given in the standard fashion as the least fixpoint of an appropriate compositionally defined operator:

A *valuation* of a context Γ is a function η such that $\eta(x) \in [\![\Gamma(x)]\!]$ for each $x \in \mathrm{dom}(\Gamma)$; a valuation of a signature Σ is a function ρ such that $\rho(f) \in [\![\Sigma(f)]\!]$ whenever $f \in \mathrm{dom}(\Sigma)$. It is valid if it interprets the constructors and destructors for lists and trees by the eponymous set-theoretic operations

To each expression e such that $\Gamma \vdash_\Sigma e : A$ we assign an element $[\![e]\!]_{\eta,\rho} \in [\![A]\!] \cup \{\bot\}$ in the obvious way, i.e. function symbols and variables are interpreted according to the valuations; basic functions and expression formers are interpreted by the eponymous set-theoretic operations, ignoring the arguments of type $\Diamond$ in the case of constructor functions. The formal definition of $[\![-]\!]_{\eta,\rho}$ is by induction on terms. A *program* $(\Sigma, (e_f)_{f \in \mathrm{dom}(\Sigma)})$ is interpreted as the least valuation ρ such that

$$\rho(f)(v_1, \ldots, v_n) = [\![e_f]\!]_{\rho,\eta}$$

where $\eta(x_i) = v_i$.

We stress that this set-theoretic semantics does not say anything about space usage. Its *only* purpose is to pin down the functional denotations of programs so that we can formally state what it means to implement a function. Accordingly, the resource type is interpreted as a singleton set and $\otimes$ product is interpreted as cartesian product.

It will be our task to show that the `malloc()`-free interpretation of our language is faithful with respect to the set-theoretic semantics. Once this is done, the user of the language can think entirely in terms of the semantics as far as extensional verification and development of programs is concerned. In addition, he or she can benefit from the resource bounds obtained from the interpretation but need not worry about how these are guaranteed.

3.3 Examples

Reverse:

$$\begin{aligned}
&\texttt{rev\_aux} : (\mathsf{L}(\mathsf{N}), \mathsf{L}(\mathsf{N})) \to \mathsf{L}(\mathsf{N})\\
&\texttt{reverse} : (\mathsf{L}(\mathsf{N})) \to \mathsf{L}(\mathsf{N})\\
&e_{\texttt{rev\_aux}}(l, acc) = \mathsf{match}\ l\ \mathsf{with}\\
&\qquad \mathsf{nil} \Rightarrow acc\\
&\qquad |\mathsf{cons}(d, h, t) \Rightarrow \texttt{rev\_aux}(t, \mathsf{cons}(d, h, acc))\\
&e_{\texttt{reverse}}(l) = \texttt{rev\_aux}(l, \mathsf{nil}_{\mathsf{N}})
\end{aligned}$$

Insertion sort

```
insert : $(\Diamond, \mathsf{N}, \mathsf{L}(\mathsf{N}))\to\mathsf{L}(\mathsf{N})$
sort : $(\mathsf{L}(\mathsf{N}))\to\mathsf{L}(\mathsf{N})$
$e_{\mathtt{insert}}(d, a, l) = \mathsf{match}\ l\ \mathsf{with}$
      $\mathsf{nil}\Rightarrow\mathsf{nil}$
      $|\mathsf{cons}(d', b, l)\Rightarrow\mathsf{if}\ a \leq b$
         $\mathsf{then}\ \mathsf{cons}(d, a, \mathsf{cons}(d', b, l))$
         $\mathsf{else}\ \mathsf{cons}(d, b, \mathtt{insert}(d', b, l))$
$e_{\mathtt{sort}}(l) = \mathsf{match}\ l\ \mathsf{with}$
      $\mathsf{nil}\Rightarrow\mathsf{nil}$
      $|\mathsf{cons}(d, a, l)\Rightarrow\mathtt{insert}(d, a, \mathtt{sort}(l))$
```

Breadth-first search

```
snoc : $(\Diamond, \mathsf{L}(\mathsf{T}(\mathsf{N})), \mathsf{T}(\mathsf{N}))\to\mathsf{L}(\mathsf{T}(\mathsf{N}))$
breadth : $(\mathsf{L}(\mathsf{T}(\mathsf{N})))\to\mathsf{L}(\mathsf{N})$
$e_{\mathtt{snoc}}(d, l, t) = \mathsf{match}\ l\ \mathsf{with}$
      $\mathsf{nil}\Rightarrow\mathsf{cons}(d, t, \mathsf{nil}())$
      $|\mathsf{cons}(d', t', q)\Rightarrow\mathsf{cons}(d', t', \mathtt{snoc}(d, q, t))$
$e_{\mathtt{breadth}}(q) = \mathsf{match}\ q\ \mathsf{with}$
      $\mathsf{nil}\Rightarrow\mathsf{nil}$
      $|\mathsf{cons}(d, t, q) = \mathsf{match}\ t\ \mathsf{with}$
         $\mathsf{leaf}(a)\Rightarrow\mathsf{cons}(d, a, \mathtt{breadth}(q))$
         $\mathsf{node}(d_1, d_2, a, l, r)\Rightarrow\mathsf{cons}(d, a,$
            $\mathtt{breadth}(\mathtt{snoc}(d_2, \mathtt{snoc}(d_1, q, l), r)))$
```

Other examples we have tried out include quicksort, treesort, and the Huffman algorithm.

Remark 31 *It can be shown that all definable functions are non-size-increasing, e.g., if $f : (\mathsf{L}(\mathsf{N}))\to\mathsf{L}(\mathsf{N})$ then, semantically, $|f(l)| \leq |l|$. This would not be the case if we would omit the $\Diamond$ argument in* cons*, even if we keep linearity. We would then, for example, have the function $f(l) = \mathsf{cons}(0, l)$ which increases the length. The presence of such a function in the body of a recursive definition gives rise to arbitrarily long lists.*

3.4 Compilation into C

By following the pattern of the examples in the introduction it is possible to associate a piece of `C`-code $[\![P]\!]^{\mathsf{C}}$ to each program $P = (\Sigma, (e_f)_{f\in\mathrm{dom}(\Sigma)})$ in such a way that

1. To each zero-order type A occurring in P a unique `C` identifier $\nu(A)$ is associated and $[\![P]\!]^{\mathsf{C}}$ contains an appropriate type definition of this identifier along with appropriately typed helper functions, e.g. $\nu(A)$`_cons`, $\nu(A)$`_list_destr` when $A = \mathsf{L}(\ldots)$.
2. For each function symbol $f : (A_1, \ldots, A_n)\to B$ defined in P the code $[\![P]\!]^{\mathsf{C}}$ contains a corresponding definition $[\![f]\!]^{\mathsf{C}}$ of a function f with prototype
$\nu(B)$ `f(`$\nu(A_1)$ x_1`, ..., `$\nu(A_n)$ x_n`)`
3. Whenever $\Gamma \vdash_{\Sigma} e : A$ then we can exhibit a `C` expression $[\![e]\!]^{\mathsf{C}}$ of type $\nu(A)$ and involving the identifiers in Γ and in Σ.

The details of this translation are omitted for lack of space; its gist is, however, contained in the examples from the introduction.

3.5 Correctness of the Translation

We now have to show that the translation $[\![P]\!]^{\mathsf{C}}$ of a program P computes the partial functions defined by the set-theoretic interpretation ρ of P. Since we have not given all details of the translation we must content ourselves with a sketch of the correctness theorem and its proof which should hopefully allow the inclined reader to reconstruct it in full.

For each zero-order type A we define the set $\mathcal{V}(A)$ as the set of pairs (v, H) where v is a C-stack-value of type $\nu(A)$ (under the type definitions $[\![P]\!]^{\mathsf{C}}$) and H is a region in the heap (a set of addresses).

For example, an element of $\mathcal{V}(\mathsf{L}(\mathsf{N}))$ consists of a stack-value of

```
typedef struct lnode {
  kind_t kind;int hd;struct lnode * tl;
} list_t;
```

i.e., a triple $v = (k, h, t)$ where k, h are (4 byte) integers and t is a memory address together with a set H of memory addresses. This set of memory addresses is meant to, but at this point not required to, comprise all addresses reachable from t by iterated dereferencing.

Next, we inductively define a relation $\Vdash_A \subseteq \mathcal{V}(A) \times [\![A]\!]$ which singles out the values which "implement" or "correspond to" a given semantic value.

- $(n, \emptyset) \Vdash_{\mathsf{N}} n'$, if n encodes n'
- $(p, H) \Vdash_{\diamond} 0$, if H is a contiguous region of size $\max\{\texttt{sizeof(}\nu(A)\texttt{)} \mid A \text{ occurs in } P\}$ and p points to the beginning of H.
- $(v, H) \Vdash_{A \otimes B} (a, b)$ if $H = H_1 \dot{\cup} H_2$ and $v\texttt{.fst}, H_1 \Vdash_A a$ and $v\texttt{.snd}, H_2 \Vdash_B b$.
- $(v, \emptyset) \Vdash_{\mathsf{L}(A)} \mathsf{nil}$ if $v\texttt{.kind} = \texttt{NIL}$.
- $(v, H) \Vdash_{\mathsf{L}(A)} \mathsf{cons}(h, t)$, if $v\texttt{.kind} = \texttt{CONS}$ and $H = H_d \dot{\cup} H_h \dot{\cup} H_t$ and $(v\texttt{.tl}, H_d) \Vdash_{\diamond} 0$ and $(v\texttt{.hd}, H_t) \Vdash_A h$ and $(v\texttt{.tl}, H_t) \Vdash_{\mathsf{L}(A)} t$,
- $(v, H) \Vdash_{\mathsf{T}(A)} \mathsf{leaf}(a)$ if $v\texttt{.kind} = \texttt{LEAF}$ and $(v\texttt{.label}, H) \Vdash_A a$,
- $(v, H) \Vdash_{\mathsf{T}(A)} \mathsf{node}(a, l, r)$ if $v\texttt{.kind} = \texttt{NODE}$ and $H = H_{d1} \dot{\cup} H_{d2} \dot{\cup} H_a \dot{\cup} H_l \dot{\cup} H_r$ and $(v\texttt{.left}, H_{d1}) \Vdash_{\diamond} 0$ and $(v\texttt{.right}, H_{d2}) \Vdash_{\diamond} 0$ and $(v\texttt{.label}, H_a) \Vdash_A a$ and $(v\texttt{.left}, H_l) \Vdash_{\mathsf{T}(A)} l$ and $(v\texttt{.right}, H_r) \Vdash_{\mathsf{T}(A)} r$

Here $H = H_1 \dot{\cup} H_2$ means that $H = H_1 \cup H_2$ and $H_1 \cap H_2 = \emptyset$.

Notice that whenever A is heap-free and $(v, H) \Vdash_A a$ for some a then $H = \emptyset$.

Theorem 32 *Assume the following:*

- *a program $P = (\Sigma, (e_f)_{f \in \mathrm{dom}(\Sigma)})$,*
- *a well typed expression $\Gamma \vdash_\Sigma e : A$,*
- *for each $x \in \Gamma$ a value $(v_x, H_x) \in \mathcal{V}(\Gamma(x))$ such that $H_x \cap H_y = \emptyset$ whenever $x \neq y$,*
- *a mapping η such that $(v_x, H_x) \Vdash_{\Gamma(x)} \eta(x)$ for each $x \in \mathrm{dom}(\Gamma)$,*

Let ρ be the set-theoretic interpretation of P.

Then the evaluation of $[\![e]\!]^{\mathsf{C}}_{[x_1 \mapsto x_1, \ldots, x_n \mapsto x_n]}$ in a runtime environment which maps $x \in \mathrm{dom}(\Gamma)$ to v_x will result in a value v such that $(v, H) \Vdash_A [\![e]\!]_{\eta,\rho}$ for some subset $H \subseteq \bigcup_{x \in \mathrm{dom}(\Gamma)} H_x$ and moreover the part of the heap outside of $\bigcup_{x \in \mathrm{dom}(\Gamma)} H_x$ will be left unaffected by the evaluation.

Proof. Straightforward lexicographic induction on evaluation time and length of typing derivations. Details are omitted for lack of space.

It follows by specialising to the defining expressions e_f that a program computes its set-theoretic interpretation.

4 Extensions

Dynamic allocation As it stands there is no way to create a value of type $\diamond$, so in particular, it is not possible to create a non-nil constant of list type. The examples show that this is often not needed. Sometimes, however, dynamic allocation and deallocation may be required and to this end we can introduce functions $\mathsf{new} : ()\to\diamond$ and $\mathsf{disp} : (\diamond)\to\mathsf{N}$. The full paper explains how these are translated and used.

Polymorphism, higher-order functions We can extend the language with polymorphism (with two kinds of type variables ranging over zero- and first order types) and higher-order functions, both linear and nonlinear. Recursive functions would then be defined using a single constant

$$\mathsf{rec} : \forall X.!(!X \multimap X) \multimap X$$

where X ranges over first-order types. The full paper contains a more detailed discussion of this point.

Queues The program for breadth-first search could be made more efficient using queues with constant time enqueuing. We can easily add a type former $\mathsf{Q}(A)$ (and appropriate term formers) which gets translated into linked lists with a pointer to their end. The correctness proof carries over with only minor changes.

Tail recursion The type system does not impose any restriction on the size of the *stack*. If a bounded stack size is desired, all we need to do is restrict to a tail recursive fragment and translate the latter into iteration.

More challenging would be some automatic program transformation which translates the existing definition of `breadth` and similar functions into iterative code. To what extent this can be done systematically remains to be seen. It seems that at least for linear recursion (only one recursive call) such transformation might always be possible using continuations.

Expressivity In order to study complexity-theoretic expressivity it seems to be a reasonable abstraction to view the type N as finite, e.g. the set of 32 bit words, and to view the heap as infinite. In this case, we have the following expressivity result:

Theorem 41 *If $f : \mathbb{N} \to \mathbb{N}$ is a non-increasing function computable in linear (in $\log(n)$) space then there exists a program containing a symbol $\mathtt{f} : (\mathsf{L}(\mathsf{N}))\to\mathsf{L}(\mathsf{N})$ such that $[\![\mathtt{f}]\!](u(x)) = u(\mathtt{f}(x))$ when $u : \mathbb{N} \to \{0,1\}^*$ is an encoding of natural numbers as lists of 0s and 1s.*

Proof. If $f(n)$ is computable in space $c\log(n)$ then we use the type $T = \mathsf{L}(\mathsf{N} \otimes \ldots \otimes \mathsf{N})$ with c factors to store memory configurations. We obtain f by iterating a one-step function of type $(T)\to T$ and composing with an initialisation function of type $(\mathsf{L}(\mathsf{N}))\to T$ and an output extraction function of type $(T)\to\mathsf{L}(\mathsf{N})$ all of which are readily seen to be implementable in our system.

If we restrict to a tail recursive fragment then programs can also be evaluated in linear space so that we obtain a characterisation of linear space.

Recursive types We can extend the type system and the compilation technique to arbitrary (even nested) first-order recursive types. To that end, we introduce (zero order) type variables and a new type former $\mu X.A$ which binds X in A. Elements of $\mu X.A$ would be introduced and eliminated using fold and unfold constructs

$$\frac{\Gamma \vdash_\Sigma e : A[(\Diamond \otimes \mu X.A)/X]}{\Gamma \vdash_\Sigma \mathsf{fold}(e) : \mu X.A} \qquad \text{(FOLD)}$$

$$\frac{\Gamma \vdash_\Sigma e : \mu X.A}{\Gamma \vdash_\Sigma \mathsf{unfold}(e) : A[(\Diamond \otimes \mu X.A)/X]} \qquad \text{(UNFOLD)}$$

. This together with coproduct and unit types allows us to *define* lists and trees as recursive datatypes. Notice that this encoding would also charge two $\Diamond$s for a tree constructor.

5 Conclusion

We have defined a linearly typed first-order language which gives the user explicit control over heap space in the form of a resource type.

A translation of this system into `malloc()`-free `C` is given which in the case of simple examples such as list reversal and quicksort generates the usual textbook solutions with in-place update.

We have shown the correctness of this compilation with respect to a standard set-theoretic semantics which disregards linearity and the resource type and demonstrated the applicability by a range of small examples.

The main selling points of the approach are

1. that it achieves in place update of heap allocated data structures while retaining the possibility of equational reasoning and induction for the verification and
2. that it generates code which is guaranteed to run in a heap of statically determined size.

This latter point should make the system interesting for applications where resources are limited, e.g. computation over the Internet, proof-carrying code, and embedded systems. Of course further work, in particular an integration with a fully-fledged functional language and the possibility of allocating a fixed amount of extra heap space will be required. Notice, however, that this latter effect can already be simulated by using input of the form $\mathsf{L}(\Diamond \otimes A)$ as opposed to $\mathsf{L}(A)$.

Also, a type inference system relieving the user from having to explicitly move around the $\Diamond$-resource might be helpful although the present system has the advantage of showing the user in an abstract and understandable way where space is being consumed. And perhaps some programmers might even enjoy spending and receiving $\Diamond$s.

6 Related Work

While the idea of translating linearly typed functional code directly into `C` seems to be new there exist a number of related approaches aimed at controlling the space usage of functional programs.

Tofte-Talpin's region calculus [19] tries to minimise garbage collection by dividing the heap into a list of regions which are allocated and deallocated according to a stack discipline. A type systems ensures that the deallocation of a region does not destroy data which is still needed; an inference system [20] generates the required annotations automatically for raw ML code.

The difference to the present work is not so much the inference mechanism (see above) but the fact that even with regions the required heap size is potentially unbounded whereas the present system guarantees that the heap will not grow. Also in place update does not take place.

Hughes and Pareto's system of sized types annotates list types with their length, e.g. the reversal function would get type $\forall n.\mathsf{L}_n(A) \to \mathsf{L}_n(A)$. While this system allows one to estimate the required heap and stack size it does not perform in place update either (and cannot due to the absence of linear types).

In a similar vein Crary and Weirich [7] have given a type system which allows one to formalise and certify informal reasoning about time consumption of recursive programs involving lists and trees. Their language is a standard one and no optimisation due to heap space reuse is taken into account.

The relationship between linear types and garbage collection has been recognised as early as '87 by Lafont [14], see also [10,1,21,16]. But again, due to the absence of $\diamond$-types, these systems do not provide in place update but merely deallocate a linear argument immediately after its use.

This effect, however, is already achieved by traditional reference counting which may be the reason why linear functional programming hasn't really got off the ground, see also [6]. While the runtime advantages of the present approach might also be realised through reference counting (and indeed seem to be by the OCAMLOPT compiler) the distinctive novelty lies in the fact that one can *guarantee* bounded heap size and obtain a simple C program realising it which can be run on any machine or system supporting C.

The type system itself is very similar to the system described by the author in [9] which in turn was inspired by Caseiro's analysis of recursive equations [5] and bears some remote similarity with Bounded Linear Logic [8]

Mention should also be made of Baker's Linear LISP [2,3] which bears some similarity to our language. It does not contain the resource type $\diamond$ or a comparable feature, thus it is not clear how the size of intermediate data structures is limited, cf. Remark 31. Similar ideas, without explicit mention of linearity are also contained in Mycroft's thesis [17]

Other related approaches are *uniqueness types* in Clean [4], linear ADTs and monads [11] which will be compared in the full paper.

In a seminar talk in Edinburgh, John Reynolds has reported about ongoing work on using linear types for in-place update. At the time of writing there was no conclusive result, though and his attention seems to have since shifted to using linear types for *reasoning* about *shared* heap allocated data structures. This together with a medium depth literature research leads me to believe that the present article is in fact the first to successfully apply linear types to the problem of functional in-place update.

Acknowledgement I would like to thank Samson Abramsky for helpful comments and encouragements. Thanks are also due to Peter Selinger for spotting a shortcoming in an earlier version of this paper.

References

1. Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.
2. Henry Baker. Lively Linear LISP—Look Ma, No Garbage. *ACM Sigplan Notices*, 27(8):89–98, 1992.
3. Henry Baker. A Linear Logic Quicksort. *ACM Sigplan Notices*, 29(2):13–18, 1994.
4. E. Barendsen and S. Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, 6:579–612, 1996.
5. Vuokko-Helena Caseiro. *Equations for Defining Poly-time Functions.* PhD thesis, University of Oslo, 1997. Available by ftp from ftp.ifi.uio.no/pub/vuokko/0adm.ps.
6. J. Chirimar, C. Gunter, and J. Riecke. Reference counting as a computational interpretation of linear logic. *Journal of Functional Programming*, 6(2), 1995.
7. K. Crary and S. Weirich. Resource bound certification. In *Proc. 27th Symp. Principles of Prog. Lang. (POPL).* ACM, 2000. to appear.
8. J.-Y. Girard, A. Scedrov, and P. Scott. Bounded linear logic. *Theoretical Computer Science*, 97(1):1–66, 1992.
9. Martin Hofmann. Linear types and non size-increasing polynomial time computation. In *Logic in Computer Science (LICS).* IEEE, Computer Society Press, 1999. to appear.
10. Sören Holmström. A linear functional language. In *Proceedings of the Workshop on Implemenation of Lazy Functional Languages.* Chalmers University, Göteborg, Programming Methodology Group, Report 53, 1988.
11. Paul Hudak and Chih-Ping Chen. Rolling your own mutable adt — a connection between linear types and monads. In *Proc. Symp. POPL '97, ACM*, 1997.
12. J. Hughes and L. Pareto. Recursion and dynamic data structures in bounded space: towards embedded ml programming. In *Proc. International Conference on Functional Programming. Paris, September '99.*, 1999. to appear.
13. Kelley and Pohl. *A book on C, third edition.* Benjamin/Cummings, 1995.
14. Yves Lafont. The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988.
15. Xavier Leroy. The Objective Caml System, documentation and user's guide. Release 2.02. http://pauillac.inria.fr/ocaml/htmlman, 1999.
16. P. Lincoln and J. Mitchell. Operational aspects of linear lambda calculus. In *Proc. LICS 1992, IEEE*, 1992.
17. Alan Mycroft. *Abstract interpretation and optimising transformations for applicative programs.* PhD thesis, Univ. Edinburgh, 1981.
18. George Necula. Proof-carrying code. In *Proc. 24th Symp. Principles of Prog. Lang. (POPL).* ACM, 1997. to appear.
19. M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.
20. Mads Tofte and Lars Birkedal. Region inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(5):724–767, 1998.
21. D. Turner and P. Wadler. Operational interpretations of linear logic. *Theoretical Computer Science*, 1999. to appear.

Secure Information Flow as Typed Process Behaviour

Kohei Honda[1], Vasco Vasconcelos[2], and Nobuko Yoshida[3]

[1] Queen Mary and Westfield College, London, U.K.
[2] University of Lisbon, Lisbon, Portugal.
[3] University of Leicester, Leicester, U.K.

Abstract. We propose a new type discipline for the π-calculus in which secure information flow is guaranteed by static type checking. Secrecy levels are assigned to channels and are controlled by subtyping. A behavioural notion of types capturing causality of actions plays an essential role for ensuring safe information flow in diverse interactive behaviours, making the calculus powerful enough to embed known calculi for type-based security. The paper introduces the core part of the calculus, presents its basic syntactic properties, and illustrates its use as a tool for programming language analysis by a sound embedding of a secure multi-threaded imperative calculus of Volpano and Smith. The embedding leads to a practically meaningful extension of their original type discipline.

1 Introduction

In present-day computing environments, a user often employs programs which are sent or fetched from different sites to achieve her/his goals, either privately or in an organisation. Such programs may be run as a code to do a simple calculation task or as interactive parallel programs doing IO operations or communications, and sometimes deal with secret information, such as private data of the user or classified data of the organisation. Similar situations may occur in any computing environments where multiple users share common computing resources. One of the basic concerns in such a context is to ensure programs do not leak sensitive data to the third party, either maliciously or inadvertently. This is one of the key aspects of the security concerns, which is often called *secrecy*. Since it is difficult to dynamically check secrecy at run-time, it may as well be verified statically, i.e. from a program text alone [7]. The *information flow analysis* [7,11,25] addresses this concern by clarifying conditions when flow of information in a program is safe (i.e. high-level information never flows into low-level channels). Recent studies [2,35,33] have shown how we can integrate the techniques of type inference in programming languages with the ideas of information flow analysis, accumulating the basic principles of compositional static verification for secure information flow.

The study of type-based secrecy so far has been done in the context of functional or imperative calculi that incorporate secrecy. Considering that concurrency and communication are a norm in modern programming environments,

G. Smolka (Ed.): ESOP/ETAPS 2000, LNCS 1782, pp. 180–199, 2000.

one may wonder whether a similar study is possible in the framework of process calculi. There are two technical reasons why such an endeavour can be interesting. First, process calculi have been accumulating mathematically rigorous techniques to reason about computation based on communicating processes. In particular, given that an equivalence on program phrases plays a basic role for semantic justification of a type discipline for secrecy [35], the theories of behavioural equivalences [17,20,26,28], which are a cornerstone in the study of process calculi, would offer a semantic basis for safe information flow in communicating processes. Second, type disciplines for communicating processes are widely studied recently, especially in the context of name passing process calculi such as the π-calculus, e.g. [6,15,20,28,32,36]. Further, recent studies have shown that name passing calculi enjoy great descriptive power, uniformly representing diverse language constructs as name passing processes, including those of sequential, concurrent, imperative, functional and object-oriented languages. Since many real-life programming languages are equipped with diverse constructs from different programming paradigms, it would be interesting to see whether we can obtain a typed calculus based on name passing in which information flow involving various language constructs are analysable on a uniform syntactic basis.

Against these backgrounds, the present work introduces a typed π-calculus in which secure information flow is guaranteed by static typing. Secrecy levels are attached to channels, and a simple subtyping ensures that interaction is always secrecy-safe. Information flow in this context arises as transformation of interactive behaviour to another interactive behaviour. Thus the essence of secure information flow becomes that a low-level interaction never depends on a high-level (or incompatible-level) interaction. Interestingly, this interaction-based principle of secure information flow strongly depends on the given type structures as *prerequisites*: that is, even semantically, certain behaviours can become either secure or insecure according to the given types. This is because types restrict a possible set of behaviours (which act as information in the present context), thus affecting the notion of safe information flow itself. For this reason, a strong type discipline for name passing processes for linear and deadlock-free interaction [6,20,36] plays a fundamental role in the present typed calculus, by which we can capture safety of information flow in a wide range of computational behaviours, including those of diverse language constructs. This expressiveness can be used to embed and analyse typed programming languages for secure information flow. In this paper we explore the use of the calculus in this direction through a sound embedding of a secure multi-threaded imperative calculus of Volpano and Smith [33]. The embedding offers an analysis of the original system in which the underlying observable scenario is made explicit and is elucidated by typed process representation. As a result, we obtain a practically meaningful extension of [33] with enlarged typability. We believe this example suggests a general use of the proposed framework, given the fundamental importance of the notion of observables in the analysis of secure computing systems [25,33,34].

Technically speaking, our work follows, on the one hand, Abadi's work on type-based secrecy in the π-calculus [1] and the studies on secure information flow in CCS and CSP [8,24,29,31], and, on the other, the preceding works on type disciplines for name passing processes. In comparison with [1], the main novelty of the present typing system is that it ensures safety of information flow for general process behaviours rather than that for ground values, which is often essential for the embedding of securely typed programming languages. Compared to [8,24,31], a key difference lies in the fundamental role type information plays in the present system for defining and guaranteeing secrecy. Further, these works are not aimed at ensuring secrecy via static typing. Other notable works on the study of security using name passing processes include [3,5]. These works are not about information flow analysis, though they do address other aspects of secrecy.

In the context of type disciplines for name passing processes, the full use of dualised and directed types (cf. §3), as well as their combination with causality-based dynamic types, is new, though the ideas are implicit in [4,10,14,20,36]. Our construction is based on graph-based types in [36], incorporating the partial algebra of types from [15] (the basic idea of modalities used here and in [15] originally comes from linear logic [10]). The syntax of the present calculus is based on [32], among others branching and recursion. We use the synchronous version since it gives a much simpler typing system. The branching and recursion play an essential role in type discipline, as we shall discuss in § 3. The calculus is soundly embeddable into the asynchronous π-calculus (also called the ν-calculus [17]) by concise encoding [32]. The operational feasibility of branching and recursion is further studied in [9,23]. For non-deterministic secrecy in general, security literature offers many studies based on probabilistic non-interference, cf. [13]. The present calculus and its theory are introduced as a basic stratum for the study of secure information flow in typed name passing processes, focussing on a simpler realm of possibilistic settings. Incorporation of the probability distribution in behavioural equivalences [22] is an important subject of future study. Further discussions on related works, including comparisons with functional and imperative secure calculi, are given in the full version [16].

This paper offers a summary of key technical ideas and results, leaving the detailed theoretical development to the full version [16]. In the remainder, Section 2 informally illustrates the basic ideas using examples. Section 3 introduces types, subtyping and the typing rules. Section 4 discusses key syntactic properties of typed terms. Finally Section 5 presents the embedding result and discusses how it suggests an extension of the original type discipline by Volpano and Smith.

Acknowledgement. We deeply thank anonymous referees for their significant comments on an early version. Our thanks also go to Martin Berger, Gavin Lowe, Peter O'Hearn, Edmund Robinson and Pasquale Malacaria for their comments and discussions.

2 Basic Ideas

2.1 A Simple Principle

Let us consider how the notion of information flow arises in interacting processes, taking a simplest example. A CCS term $a.\overline{b}.\mathbf{0}$ represents a behaviour which synchronises at a as input, then synchronises at b as output, and does nothing. Suppose we attach a secrecy level to each port, for example "High" to a and "Low" to b. Intuitively this means that we wish interaction at a to be secret, while interaction at b may be known by a wider public: any high-level security process may interact at a and b, while a low-level security process can interact only at b. Then this process represents insecure interactions: any process observing b, which can be done by a low-level process, has the possibility to know an interaction at a, so information is indeed transmitted to a lower level from a higher level. Note that this does not depend on a being used for input and b used for output: $\overline{a}.b.\mathbf{0}$ with the same assignment of secrecy levels is similarly unsafe. In both cases, we are saying that if there is a causal dependency from an action at a high-level channel to the one at a low-level channel, the behaviour is not safe from the viewpoint of information flow. Further, if we have value passing in addition, we would naturally take dependency in terms of communicated values into consideration.

The above informal principle based on causal dependency[1] is simple, but may look basic as a way of stipulating information flow for processes. Since many language constructs are known to be representable as interacting processes [18,19], one may wonder whether the above idea can be used for understanding safety in information flow in various programming languages. In the following, we consider this question by taking basic examples of information flow in imperative programs.

2.2 Syntax

Let $a, b, c, \ldots x, y, z, \ldots$ range over *names* (which are both points of interaction and values to be communicated), and $X, Y, \ldots$ over *agent variables.* We write $\vec{y}$ for a vector of names $y_0 \cdots y_{n-1}$ with $n \geq 0$. Then the syntax for *processes*, written $P, Q, R, \ldots$, is given by the following grammar. We note that this syntax extends the standard polyadic π-calculus with branching and recursion. These extensions play a fundamental role in the type discipline, in that intended types are hard to deduce if we use their encoding into, say, the polyadic π-calculus (see [16] for further discussions).

$$\begin{array}{rlll}
P ::= & x(\vec{y}).P & \text{input} & \mid P \mid Q \quad \text{parallel} \\
\mid & \overline{x}\langle(\boldsymbol{\nu}\,\vec{z})\vec{y}\rangle.P & \text{output} & \mid (\boldsymbol{\nu}\,x)P \quad \text{hiding} \\
\mid & x[(\vec{y}).P \,\&\, (\vec{z}).Q] & \text{branching input} & \mid \mathbf{0} \quad \text{inaction} \\
\mid & \overline{x}\,\mathtt{inl}\langle(\boldsymbol{\nu}\,\vec{z})\vec{y}\rangle.P & \text{left selection} & \mid X\langle\vec{x}\rangle \quad \text{recursive variable} \\
\mid & \overline{x}\,\mathtt{inr}\langle(\boldsymbol{\nu}\,\vec{z})\vec{y}\rangle.P & \text{right selection} & \mid (\mu X(\vec{x}).P)\langle\vec{y}\rangle \quad \text{recursion}
\end{array}$$

[1] Related ideas are studied in the context of CCS [8] and CSP [31].

There are two kinds of inputs, one unary and another binary: the former is the standard input in the π-calculus, while the latter, the *branching input*, has two branches, waiting for one of them to be selected with associated communication [32]. Accordingly there are outputs with left and right selections, as well as the standard one. We require all vectors of names in round parenthesis are pairwise distinct, which act as binders. In the value part of an output (including selections), say $\langle(\boldsymbol{\nu}\,\vec{z})\vec{y}\rangle$, names in $\vec{z}$ should be such that $\{\vec{z}\} \subset \{\vec{y}\}$ ($\{\vec{x}\}$ is the set of names in $\vec{x}$), and the order of occurrences of names in $\vec{z}$ should be the same as the corresponding names in $\vec{y}$. Here $(\boldsymbol{\nu}\,\vec{z})$ indicate names $\vec{z}$ are new names and are exported by output. $\langle(\boldsymbol{\nu}\,\vec{z})\vec{y}\rangle$ is written $\langle\vec{y}\rangle$ if $\vec{z} = \emptyset$, and $(\boldsymbol{\nu}\,\vec{z})$ if $\vec{y} = \vec{z}$. We often omit vectors of the length zero (for example, we write $\mathtt{inr}$ for $\mathtt{inr}\langle\ \rangle$) as well as the trailing $\mathbf{0}$. The binding and α-convertibility $\equiv_\alpha$ are defined in the standard way. In a recursion $(\mu X(\vec{x}).P)\langle\vec{y}\rangle$, we require that P is *input guarded*, that is P is either a unary input or a branching input, and free names in P are a subset of $\{\vec{x}\}$. The reduction relation $\longrightarrow$ is defined in the standard manner, which we illustrate below (the formal definition is given in [16]).

We illustrate the syntax by examples. First, the following agents represent boolean constants denoting the truth and the conditional selection (let c and y be fresh).

$$\mathbf{T}\langle b\rangle \;=\; b(c).(\overline{c}\,\mathtt{inl} \mid \mathbf{T}\langle b\rangle) \qquad \text{and} \qquad \mathbf{If}\langle x,\ P,\ Q\rangle \overset{\text{def}}{=} \overline{x}(\boldsymbol{\nu}\,y).y[().P\,\&\,().Q]$$

The recursive definition of $\mathbf{T}\langle b\rangle$ is a notational convention and actually stands for $\mathbf{T}\langle b\rangle \overset{\text{def}}{=} (\mu X(b).b(c).(\overline{c}\mathtt{inl} \mid X\langle b\rangle))\langle b\rangle$. The truth agent first inputs a name c via b, then, via c, does the left selection with no value passing as well as recreating the original agent. By replacing $\mathtt{inl}$ by $\mathtt{inr}$, we can define the falsity. The conditional process invokes a boolean agent, then waits with two branches. If the other party is truth it generates P: if else it generates Q. We can now show how these two processes interact:

$$\mathbf{If}\langle x,\ P,\ Q\rangle \mid \mathbf{T}\langle x\rangle \;\longrightarrow\; (\boldsymbol{\nu}\,y)(y\,[().P\,\&\,().Q] \mid \overline{y}\,\mathtt{inl} \mid \mathbf{T}\langle x\rangle) \;\longrightarrow\; P \mid \mathbf{T}\langle x\rangle$$

Next we consider a representation of imperative variable as a process.

$$\mathbf{Var}\langle xv\rangle = x[(z).(\overline{z}\langle v\rangle \mid \mathbf{Var}\langle xv\rangle)\ \&\ (v').\mathbf{Var}\langle xv'\rangle]$$

In this representation, we label the main interaction point of the process (called *principal port* in Interaction Net [21]) by the name of the variable x. It has two branches, of which the left one corresponds to the "read" option, while the right one corresponds to the "write" option. If the "read" is selected and z is received, the process sends the current value v to z, while regenerating the original self. On the other hand, if the "write" branch is selected and v' is received, then the process regenerates itself with a new value v'. We can then consider the representation of the assignment "$x := y$," which first "reads" the value from the variable y, then "writes" that value to the variable x.

$$\mathbf{Assign}\langle xy\rangle \overset{\text{def}}{=} \overline{y}\,\mathtt{inl}(\boldsymbol{\nu}\,z).z(v).\overline{x}\,\mathtt{inr}\langle v\rangle$$

2.3 Imperative Information Flow in Process Representation

(1) Causal Dependency. We can now turn to the information flow. We first consider the process representation of the following obviously insecure code [25].

$$x^{\mathrm{L}} := y^{\mathrm{H}}$$

Here the superscripts "L" and "H" indicate the secrecy levels of variables: thus y is a high (or secret) variable and x is a low (or public) variable. This command is insecure intuitively because the content of a secret variable becomes visible to the public through x. Following the previous discussion, its process representation becomes:

$$\mathbf{Assign}\langle x^{\mathrm{L}} y^{\mathrm{H}}\rangle \overset{\mathrm{def}}{=} \overline{y}^{\mathrm{H}}\,\mathtt{inl}(\boldsymbol{\nu}\,c).c^{\mathrm{H}}(v).\,\overline{x}^{\mathrm{L}}\,\mathtt{inr}\langle v\rangle.\ \text{nteractional}$$

Note we are labeling *channels* by secrecy levels. We can easily see that this process violates the informal principle stipulated in §2.1, because its low-level behaviour (at x) depends on its preceding high-level behaviour (at y, c). Thus this example does seem explainable from our general principle. Similarly, we can check the well-known example of implicit insecure flow "$\mathtt{if}\ z^{\mathrm{H}}\ \mathtt{then}\ x^{\mathrm{L}} := y^{\mathrm{L}}\ \mathtt{end}$" (where the information stored in z can be indirectly revealed by reading x), is translated into insecure process interaction "$\overline{z}^{\mathrm{H}}(\boldsymbol{\nu}\,c).c^{\mathrm{H}}[().\mathbf{Assign}\langle x^{\mathrm{L}} y^{\mathrm{L}}\rangle\ \&\ ().\mathbf{0}]$". Here again the low-level interactions (in $\mathbf{Assign}\langle x^{\mathrm{L}} y^{\mathrm{L}}\rangle$) depend on the high-level interactions at z and c.

(2) Deadlock-Freedom. So far there has been no difficulty in applying our general principle to process presentation of imperative information flow. However there are subtleties to be understood, one of which arises in the following sequential composition.

$$x^{\mathrm{H}} := y^{\mathrm{H}}\ ;\ z^{\mathrm{L}} := w^{\mathrm{L}}$$

The whole command is considered to be safe since whatever the content of x and y would be, they do not influence the content of z and w. However the following process representation of this command seems *not* safe in the light of our principle:

$$\overline{y}^{\mathrm{H}}\,\mathtt{inl}(\boldsymbol{\nu}\,c_1).c_1^{\mathrm{H}}(v_1).\,\overline{x}^{\mathrm{H}}\,\mathtt{inr}\langle v_1\rangle.\,\overline{w}^{\mathrm{L}}\,\mathtt{inl}(\boldsymbol{\nu}\,c_2).c_2^{\mathrm{L}}(v_2).\,\overline{z}^{\mathrm{L}}\,\mathtt{inr}\langle v_2\rangle \qquad (\star)$$

Here the behaviours at low-level ports (w and z) depend on, via prefixing, those at high-level ports (x and y). Does this mean our principle and the standard idea in information flow are incompatible with each other? However, a closer look at the above representation reveals that this problematic dependency does not exist in effect, *provided* that the above process interacts with the processes for imperative variables given in §2.2. If we assume so, the actions at y and x (together with those at z and w) by the above process are always enabled: whenever a program wishes to access a variable, it always succeeds (in the i parlance, we are saying that interactions at these names are guaranteed to be *deadlock-free*). Thus we can guarantee that, under the assumption, the action at say w above will surely take place, which means the dependency as expressed in syntax does not exist. Observing there is no dependency at the level of communicated values between the two halves of ($\star$), we can now conclude that the actions at w and z do *not* causally depend on the preceding actions at y and x.

(3) Innocuous Interaction. We now move to another subtle example, using the following command.

$$\texttt{if}\ z^{\mathrm{H}}\ \texttt{then}\ x^{\mathrm{H}} := y^{\mathrm{L}}\ \texttt{end}$$

While this phrase is considered to be secrecy-wise safe [25], its representation in the π-calculus becomes:

$$\overline{z}^{\mathrm{H}}(\boldsymbol{\nu}\, c^{\mathrm{H}}).c[().\overline{y}^{\mathrm{L}}\texttt{inl}(\boldsymbol{\nu}\, e).e^{\mathrm{L}}(v).\overline{x}^{\mathrm{H}}\,\texttt{inr}\langle v\rangle\ \&\ ().\mathbf{0}] \qquad (\star\star)$$

which again shows apparently unsafe dependency between the second action at c and the third action at y. In this example, the process does get information at c in the form of binary selection, even though c is deadlock-free. Moreover the output at y does not occur in the right branch, so the output depends on the action at c even observationally. But the preceding study [33,35] shows the original imperative behaviour is indeed safe. How can it be so? Simple, because this command only *reads* from y, without writing anything: so it is as if it did nothing to y. Returning to ($\star\star$), we find the idea we made resort to in (2), is again effective: we consider this output action as not affecting the environment (hence not transmitting any information) *provided* that the behaviour of the environment is such that invoking its left branch has no real effect – in other words, if it behaves just as the imperative variable given in §2.2 does. We call such an output *innocuous*: thus, if we decide to ignore the effect of innocuous actions, there is no unsafe dependency from the high-level to the low-level (note the left branch as a whole now becomes high-level). We further observe that the insecure examples in (1) are still insecure even after incorporating deadlock-freedom and innocuousness.

The preceding discussions suggest two things: first, we may be able to formally stipulate the interactional framework of safe information flow which may have wide applicability along the line of the informal notion given in §2.1. Secondly, however, just for that purpose, we need a non-trivial notion of types for behaviours which in particular concerns not only the behaviour of the process but also that of the assumed environment. The formal development in the following sections shows how these ideas can be materialised as a typed process calculus for safe information flow.

3 A Typed π-Calculus for Secure Information Flow

3.1 Overview

In addition to *names* and *agent variables* (cf. §2.1), the typed calculus we introduce below uses a set of multiple *secrecy levels*, which are assumed to form a lattice. $s, s', \ldots$ range over secrecy levels, and $s \leq s'$ etc. denotes the partial order (where the lesser means the lower, i.e. more public). Using these data as base sets, our objective in this section is to introduce a typing system whose provable sequent has the following form:

$\Gamma \vdash_s P \triangleright A$ a process P has an action type A under a base Γ with a secrecy level s

We offer an overview of the four elements in the above sequent.

(1) The *base* Γ is a finite function from names and agent variables to types and vectors of types, respectively. Intuitively a type assigned to a channel denotes the basic structure of *possible interaction* at that channel, for example input/output and branching/selection. We also include refined modalities for recursive inputs and their dual outputs, which indicate whether they involve state change or not.

(2) The *process* P is an untyped term in §2.2 which is annotated with types in its bound names, e.g. a unary input becomes $x(\vec{y}\,{:}\,\vec{\alpha}).P$ (here and elsewhere we assume $len(\vec{\alpha}) = len(\vec{y})$ where $len(\vec{y})$ denotes the length of a vector, so that each y_i is assigned a type α_i). As one notable aspect, we only use those processes whose outputs (in any of three forms) are *bound*, e.g. each unary output has a form $\overline{x}(\boldsymbol{\nu}\,\vec{y}\,{:}\,\vec{\alpha}).P$ (this restricted output is an important mode of communication which arises in the context of both π-calculus [30] and games semantics [19,18]). Accordingly we set names in each vector instantiating agent variables to be pairwise distinct. These restrictions make typing rules simpler, while giving enough descriptive power to serve our present purpose.

(3) The *secrecy index* s guarantees that P under Γ only affects the environment at levels at s or higher: that is, it is only transmitting information (or *tampering* the environment) at levels no less than s.

(4) The *action type* A gives abstraction of the causal dependency among (actions on) free channels in P, ensuring, among others, certain deadlock-free properties on its linear and recursive channels. The activation ordering is represented by a partial order on nodes whose typical form is $\mathsf{p}x$ where p denotes a type of action to be done at x. There is a partial algebra over action types [15], by which we can control the composability of two action types (hence of typed processes which own them), thus enabling us to stipulate assumptions on the possible forms of the environments, cf. §2.

3.2 Types and Subtyping

We start with the set of *action modes*, denoted $m, m', ...,$ whose underlying operational ideas are illustrated by the following table.

$\Downarrow$ non-linear (non-deterministic) input	$\Uparrow$ non-linear (non-deterministic) output
$\downarrow$ truly linear input (truly once)	$\uparrow$ truly linear output (truly once)
! recursive input (always available)	**?** zero or more output (always enabled)

The notations **!** and **?** come from Linear Logic [10], which first introduced these modalities. We also let $\kappa, \kappa', \ldots,$ called *mutability indices*, range over $\{\boldsymbol{\iota}, \boldsymbol{\mu}\}$.

(Well-formedness and Compatibility)

$$\frac{-}{\vdash \tau} \quad \frac{\vdash \tau \asymp \tau'}{\vdash \langle \tau,\ \tau' \rangle} \quad \frac{\vdash \tau \asymp \tau'}{\vdash \tau' \asymp \tau} \quad \frac{\vdash \tau_i \asymp \tau'_i}{\vdash (\vec{\tau})^{\Downarrow}_{s} \asymp (\vec{\tau}')^{\Uparrow}_{s}} \quad \frac{\vdash \tau_i \asymp \tau'_i \quad s \geq s'}{\vdash (\vec{\tau})^{\downarrow}_{s} \asymp (\vec{\tau}')^{\uparrow}_{s'}} \quad \frac{\vdash \tau_i \asymp \tau'_i \quad s \geq s'}{\vdash (\vec{\tau})^{\mathbf{!}}_{s,\kappa} \asymp (\vec{\tau}')^{\mathbf{?}}_{s',\kappa}}$$

$$\frac{\vdash \tau_{ij} \asymp \tau'_{ij}}{\vdash [\vec{\tau}_1 \& \vec{\tau}_2]^{\Downarrow}_{s} \asymp [\vec{\tau}'_1 \oplus \vec{\tau}'_2]^{\Uparrow}_{s}} \quad \frac{\vdash \tau_{ij} \asymp \tau'_{ij} \quad s \geq s'}{\vdash [\vec{\tau}_1 \& \vec{\tau}_2]^{\downarrow}_{s} \asymp [\vec{\tau}'_1 \oplus \vec{\tau}'_2]^{\uparrow}_{s'}} \quad \frac{\vdash \tau_{ij} \asymp \tau'_{ij} \quad s \geq s'}{\vdash [\vec{\tau}_1 \& \vec{\tau}_2]^{\mathbf{!}}_{s,\kappa_1 \& \kappa_2} \asymp [\vec{\tau}'_1 \oplus \vec{\tau}'_2]^{\mathbf{?}}_{s',\kappa_1 \oplus \kappa_2}}$$

(Subtyping)

$$\frac{\vdash \tau_i \leq \tau'_i}{\vdash (\vec{\tau})^{\Downarrow}_{s} \leq (\vec{\tau}')^{\Downarrow}_{s}} \quad \frac{\vdash \tau_i \leq \tau'_i \quad s \geq s'}{\vdash (\vec{\tau})^{\downarrow}_{s} \leq (\vec{\tau}')^{\downarrow}_{s'}} \quad \frac{\vdash \tau_i \leq \tau'_i \quad s \geq s'}{\vdash (\vec{\tau})^{\mathbf{!}}_{s,\kappa} \leq (\vec{\tau}')^{\mathbf{!}}_{s',\kappa}}$$

$$\frac{\vdash \tau_i \leq \tau'_i}{\vdash (\vec{\tau})^{\Uparrow}_{s} \leq (\vec{\tau}')^{\Uparrow}_{s}} \quad \frac{\vdash \tau_i \leq \tau'_i \quad s \leq s'}{\vdash (\vec{\tau})^{\uparrow}_{s} \leq (\vec{\tau}')^{\uparrow}_{s'}} \quad \frac{\vdash \tau_i \leq \tau'_i \quad s \leq s'}{\vdash (\vec{\tau})^{\mathbf{?}}_{s,\kappa} \leq (\vec{\tau}')^{\mathbf{?}}_{s',\kappa}}$$

$$\frac{\vdash \tau_{ij} \leq \tau'_{ij}}{\vdash [\vec{\tau}_1 \& \vec{\tau}_2]^{\Downarrow}_{s} \leq [\vec{\tau}'_1 \& \vec{\tau}'_2]^{\Downarrow}_{s}} \quad \frac{\vdash \tau_{ij} \leq \tau'_{ij} \quad s \geq s'}{\vdash [\vec{\tau}_1 \& \vec{\tau}_2]^{\downarrow}_{s} \leq [\vec{\tau}'_1 \& \vec{\tau}'_2]^{\downarrow}_{s'}} \quad \frac{\vdash \tau_{ij} \leq \tau'_{ij} \quad s \geq s'}{\vdash [\vec{\tau}_1 \& \vec{\tau}_2]^{\mathbf{!}}_{s,\kappa_1 \& \kappa_2} \leq [\vec{\tau}'_1 \& \vec{\tau}'_2]^{\mathbf{!}}_{s',\kappa_1 \& \kappa_2}}$$

$$\frac{\vdash \tau_{ij} \leq \tau'_{ij}}{\vdash [\vec{\tau}_1 \oplus \vec{\tau}_2]^{\Uparrow}_{s} \leq [\vec{\tau}'_1 \oplus \vec{\tau}'_2]^{\Uparrow}_{s}} \quad \frac{\vdash \tau_{ij} \leq \tau'_{ij} \quad s \leq s'}{\vdash [\vec{\tau}_1 \oplus \vec{\tau}_2]^{\uparrow}_{s} \leq [\vec{\tau}'_1 \oplus \vec{\tau}'_2]^{\uparrow}_{s'}} \quad \frac{\vdash \tau_{ij} \leq \tau'_{ij} \quad s \leq s'}{\vdash [\vec{\tau}_1 \oplus \vec{\tau}_2]^{\mathbf{?}}_{s,\kappa_1 \oplus \kappa_2} \leq [\vec{\tau}'_1 \oplus \vec{\tau}'_2]^{\mathbf{?}}_{s',\kappa_1 \oplus \kappa_2}}$$

$$\frac{\vdash \langle \tau_1,\ \tau_2 \rangle \quad \vdash \tau \leq \tau_1 \text{ or } \vdash \tau \leq \tau_2}{\vdash \tau \leq \langle \tau_1,\ \tau_2 \rangle} \quad \frac{\vdash \langle \tau'_1,\ \tau'_2 \rangle \quad \vdash \tau_i \leq \tau'_i}{\vdash \langle \tau_1,\ \tau_2 \rangle \leq \langle \tau'_1,\ \tau'_2 \rangle}$$

Fig. 1. Subtyping

Mutability indices indicate whether a recursive behaviour is stateful or not: for input, $\boldsymbol{\iota}$ denotes the lack of state, which we call *innocence*, cf. [19], while $\boldsymbol{\mu}$ means it may be stateful, that is it may change behaviour after invocation; for output, $\boldsymbol{\iota}$ denotes innocuousness, that is the inputting party is innocent, while $\boldsymbol{\mu}$ denotes possible lack of innocuousness. Given these base sets, the grammar of *types*, denoted $\alpha, \beta, \ldots$, are given by:

$$\begin{array}{lll}
\alpha ::= \tau \ \mid \ \langle \tau,\ \tau' \rangle & & \tau \ ::= \ \alpha_{\mathtt{I}} \ \mid \ \alpha_{\mathtt{O}} \\
\alpha_{\mathtt{I}} ::= (\vec{\tau})^{\Downarrow}_{s} \ \mid \ (\vec{\tau})^{\downarrow}_{s} \ \mid \ (\vec{\tau})^{\mathbf{!}}_{s,\kappa} \ \mid \ [\vec{\tau}_1 \& \vec{\tau}_2]^{\Downarrow}_{s} \ \mid \ [\vec{\tau}_1 \& \vec{\tau}_2]^{\downarrow}_{s} \ \mid \ [\vec{\tau}_1 \& \vec{\tau}_2]^{\mathbf{!}}_{s,\kappa_1 \& \kappa_2} & & \\
\alpha_{\mathtt{O}} ::= (\vec{\tau})^{\Uparrow}_{s} \ \mid \ (\vec{\tau})^{\uparrow}_{s} \ \mid \ (\vec{\tau})^{\mathbf{?}}_{s,\kappa} \ \mid \ [\vec{\tau}_1 \oplus \vec{\tau}_2]^{\Uparrow}_{s} \ \mid \ [\vec{\tau}_1 \oplus \vec{\tau}_2]^{\uparrow}_{s} \ \mid \ [\vec{\tau}_1 \oplus \vec{\tau}_2]^{\mathbf{?}}_{s,\kappa_1 \oplus \kappa_2} & &
\end{array}$$

Types of form $\langle \tau,\ \tau' \rangle$ are *pair types*, indicating structures of interaction for both input and output, while others are *single types*, which are only for either input or output. We write $\mathsf{md}(\alpha)$ for the set of action modes of the outermost type(s) in α, e.g. $\mathsf{md}((\vec{\tau})^{m}_{s}) = \{m\}$ and $\mathsf{md}(\langle (\vec{\tau}_1)^{m_1}_{s_1}, (\vec{\tau}_2)^{m_2}_{s_2} \rangle) = \{m_1, m_2\}$. We often write $\mathsf{md}(\alpha) = m$ for $\mathsf{md}(\alpha) = \{m\}$. Similarly, we write $\mathsf{sec}(\tau)$ for the security level of the outermost type in τ, e.g. $\mathsf{sec}((\vec{\tau})^{m}_{s}) = s$. We define the *dual* of m, written

$\overline{m}$, as: $\overline{\Downarrow} = \Uparrow$, $\overline{\Uparrow} = \Downarrow$, $\overline{\uparrow} =\downarrow$, $\overline{\downarrow} =\uparrow$, $\overline{\mathbf{!}} = \mathbf{?}$ and $\overline{\mathbf{?}} = \mathbf{!}$. Then the dual of a type α, denoted by $\overline{\alpha}$, is given by inductively dualising each action mode in α, as well as exchanging & and $\otimes$. Among types, those with body $(\vec{\tau})$ correspond to unary input/output, those with body $[\vec{\tau}_1 \& \vec{\tau}_2]$ correspond to branching input, and those with body $[\vec{\tau}_1 \oplus \vec{\tau}_2]$ correspond to output with selections.

We say α is *well-formed*, written $\vdash \alpha$, if it is derivable from the rules in Figure 1, where we also define the *compatibility relation* $\asymp$ over single types. A pair type is well-formed iff its constituting single types are compatible. We also say α is a *subtype of* β, denoted $\vdash \alpha \leq \beta$, if this sequent is derivable by the rules in Figure 1. Some comments on types, subtyping and compatibility follow.

Remark 1. **(nested types)** Nested types denote what the process would do after exporting or importing new channels (hence covariance of subtyping on nested types): as an example, neglecting the secrecy and mutability, $x : (()^{\downarrow})^{\uparrow}$ denotes the behaviour of doing a truly linear output at x exporting one single new name, and at that name doing a truly linear input without importing any name.

(secrecy levels, compatibility and subtyping) Since safe information flow should never go from a higher level to a lower level, a rule of thumb is that two types are compatible if such a flow is impossible. Thus, because a flow can occur in both ways at non-deterministic channels (cf. §2.1), two non-linear types can be related only when they have the same secrecy level. On the other hand, for compatibility of linear types, we require that the inputting side is higher than the outputting side in secrecy levels, since the flow never comes from the inputting party (further, in truly linear unary types, even the outputting party does not induce flow). Accordingly, the subtyping is covariant for output and contravariant for input with respect to secrecy levels.

(mutability index) As we explained already, the index ι represents the recursive input behaviour without state change (innocence) or, dually, the output which does not tamper the corresponding recursive processes (innocuousness). Note an index is only meaningful for recursive behaviours and their dual output. Naturally we stipulate that an innocent input can only be compatible with an innocuous output; and an innocent input can only be a subtype of an innocent input, and an innocuous output can only be a subtype of an innocuous output.

3.3 Action Types

An *action type A* is a finite poset whose elements, called *action nodes*, are given by the following grammar.

$$\mathbf{n} \ ::= \ \downarrow x \ \mid \ \uparrow x \ \mid \ \updownarrow x \ \mid \ \mathbf{!}x \ \mid \ \mathbf{?}x \ \mid \ \mathbf{?}^{\iota}x \ \mid \ \Updownarrow x \ \mid \ X\langle\vec{x}\rangle.$$

$\updownarrow x$ indicates x is already used exactly once for both input and output. $\mathbf{?}^{\iota}x$ indicates that all actions occurring at x so far are innocuous. $X\langle\vec{x}\rangle$ (with $len(\vec{x}) \geq 1$ always) indicates the point to which the behaviour recurs. $\Updownarrow$ indicates possibility of nonlinear (nondeterministic) input and output. Other symbols are already explained in the table in §3.2. As an illustration of causality, write $\mathbf{n} \to \mathbf{n}'$ when $\mathbf{n}'$ is strictly bigger than $\mathbf{n}$ without any intermediate element. Then $\downarrow x \to \uparrow y$ says that a truly linear output at y becomes active just after a truly linear input at x.

We only use those action types which conform to a well-formedness condition that in particular includes linearity (for details see [16]). In the typing rules, we use the following abbreviations for action types (let $\{x_i\}$ be free names in A).

$\downarrow\uparrow A$	A only contains $\downarrow x_i$ or $\uparrow x_i$	$A^{\text{-}x}$	x does not occur in A
$\mathbf{?}A$	A only contains $\mathbf{?}x_i$, $\mathbf{?}^{\iota}x_i$ or $\Updownarrow x_i$	$A \otimes B$	disjoint union, with $A \cap B = \emptyset$
$\mathbf{?}^{\iota}A$	A only contains $\mathbf{?}^{\iota}x_i$	$\vec{\mathtt{p}x}$	$\mathtt{p}_0 x_0 \otimes \mathtt{p}_1 x_1 \cdots \mathtt{p}_{n-1} x_{n-1}$ $(n \geq 0)$

We also say x is *active in* A if $\mathtt{p}x$ (for some $\mathtt{p}$) is minimal in A.

3.4 Typing System

We now introduce the main typing rules with illustration. We use the following notation: given a base Γ, (1) $x : \alpha$ (resp. $X : \vec{\alpha}$) denotes $\Gamma(x) = \alpha$ (resp. $\Gamma(X) = \vec{\alpha}$); and (2) $\Gamma \cdot \Delta$ denotes the disjoint union of two bases, assuming their domains do not intersect. Henceforth we assume all types and bases are well-formed. We start from the typing rules for basic process operators: the inaction, parallel composition and hiding.

$$\text{(Zero)}\ \frac{}{\Gamma \vdash_s \mathbf{0} \triangleright \emptyset} \qquad \text{(Par)}\ \frac{A_1 \asymp A_2 \quad \Gamma \vdash_s P_i \triangleright A_i \quad (i = 1, 2)}{\Gamma \vdash_s P_1 \mid P_2 \triangleright A_1 \odot A_2} \qquad \text{(Res)}\ \frac{\Gamma \cdot x : \alpha \vdash_s P \triangleright A \otimes \mathtt{p}x \quad \mathtt{p} \in \{\updownarrow, \mathbf{!}, \Updownarrow\}}{\Gamma \vdash_s (\boldsymbol{\nu}\, x : \alpha)P \triangleright A}$$

In (Par), we use *coherence* $A_1 \asymp A_2$ and *composition* $A_1 \odot A_2$, both following [36]. Essentially speaking, $A_1 \asymp A_2$ says A_1 and A_2 are composable without violating linearity or causing vicious circles; then $A_1 \odot A_2$ is the result of the composition. See [16] for details. In (Res), we do not allow a name with a mode in $\{\downarrow, \uparrow, \mathbf{?}, \mathbf{?}^{\iota}\}$ to be restricted since these actions expect their complementary actions to get composed — in other words, actions with these types assume the existence of actions with their dual types in the environment. With the complementary actions left uncomposed, the hiding leads to an insecure system. In addition, we have the weakening rules for $\mathbf{?}x$, $\mathbf{?}^{\iota}x$, $\updownarrow x$ and $\Updownarrow x$, and the *degradation rule* in which $\Gamma \vdash_s P \triangleright A$ is degraded into $\Gamma \vdash_{s'} P \triangleright A$ when $s' \leq s$ (cf. § 3.1 (3)).

We next turn to non-liner prefix rules. The rules for prefix actually control the secrecy levels of each action.

$$\text{(In)}\ \frac{\vdash (\vec{\tau})^{\Downarrow}_{s} \leq \Gamma(x) \quad \Gamma \cdot \vec{y} : \vec{\tau} \vdash_s P \triangleright \overrightarrow{\mathtt{p}y} \otimes \mathbf{?}A \otimes \Updownarrow x}{\Gamma \vdash_s x(\vec{y} : \vec{\tau}).P \triangleright A \otimes \Updownarrow x} \qquad \text{(Out)}\ \frac{\vdash (\vec{\tau})^{\Uparrow}_{s} \leq \Gamma(x) \quad \Gamma \cdot \vec{y} : \vec{\tau} \vdash_s P \triangleright \overrightarrow{\mathtt{p}y} \otimes \mathbf{?}A \otimes \Updownarrow x}{\Gamma \vdash_s \overline{x}(\boldsymbol{\nu}\, \vec{y} : \vec{\alpha}).P \triangleright A \otimes \Updownarrow x}$$

Since the subtyping on non-linear types is trivial with respect to their secrecy levels, $\vdash (\vec{\tau})^{\Uparrow, s} \leq \Gamma(x)$ means $\Gamma(x)$ has precisely the level s. Thus, in both rules, the initial action at level s is followed by actions affecting the same or higher levels (because P is typed with s). Note also all abstracted actions ($\overrightarrow{\mathtt{p}y}$ above) should be active, which is essential for the subject reduction. Non-linear prefix rules for branching and selections are essentially the same.

Among linear prefix rules, the following shows a stark contrast with the non-linear (In) and (Out) rules.

$$\begin{array}{ll}
(\mathsf{In}^{\downarrow})\quad (\text{where } C/\vec{y} = \downarrow\!\!\uparrow B) & (\mathsf{Out}^{\uparrow})\quad (\text{where } C/\vec{y} = \downarrow\!\!\uparrow B) \\
\dfrac{\begin{array}{l}\vdash (\vec{\tau})^{\downarrow}_{s'} \leq \Gamma(x) \\ \Gamma \cdot \vec{y}{:}\vec{\tau} \vdash_s P \triangleright \mathbf{?}A \otimes C^{\text{-}x}\end{array}}{\Gamma \vdash_s x(\vec{y}{:}\vec{\tau}).P \triangleright A \otimes \downarrow x \rightarrow B} & \dfrac{\begin{array}{l}\vdash (\vec{\tau})^{\uparrow}_{s'} \leq \Gamma(x) \\ \Gamma \cdot \vec{y}{:}\vec{\tau} \vdash_s P \triangleright \mathbf{?}A \otimes C^{\text{-}x}\end{array}}{\Gamma \vdash_s \overline{x}(\boldsymbol{\nu}\,\vec{y}{:}\vec{\tau}).P \triangleright A \otimes \uparrow x \rightarrow B}
\end{array}$$

The notation $C/\vec{y}$ denotes the result of taking off nodes with names among $\vec{y}$, as well as stipulating the condition that each y_i should be active in C. We observe that the "true linearity" in these and later rules is stronger than those studied in [15,20], which only requires "no more than once". In the rule, since s' is not given any condition in the antecedent, both rules completely neglect the secrecy level of x in Γ, saying we may not regard these actions as either receiving or giving information from/to the environment. The operation $\mathbf{n} \rightarrow B$, which is given in [16] following [36], records the causality.

The next rules show that branching/selection need a different treatment from the unary cases when types are truly linear. Intuitively, the act of selection gives rise to a non-trivial flow of information.

$$\begin{array}{ll}
(\mathsf{Bra}^{\downarrow})\quad (\text{where } C_i/\vec{y}_i = \downarrow\!\!\uparrow B) & (\mathsf{Sel}^{\uparrow}_{l})\quad (\text{where } C/\vec{y}_1 = \downarrow\!\!\uparrow B) \\
\dfrac{\begin{array}{l}\vdash [\vec{\tau}_1 \& \vec{\tau}_2]^{\downarrow}_{s} \leq \Gamma(x) \\ \Gamma \cdot \vec{y}_i : \vec{\tau}_i \vdash_s P_i \triangleright \mathbf{?}A \otimes C_i^{\text{-}x} \quad (i = 1,2)\end{array}}{\Gamma \vdash_s x[(\vec{y}_1 : \vec{\tau}_1).P_1 \,\&\, (\vec{y}_2 : \vec{\tau}_2).P_2] \triangleright A \otimes \downarrow x \rightarrow B} & \dfrac{\begin{array}{l}\vdash [\vec{\tau}_1 \oplus \vec{\tau}_2]^{\uparrow}_{s} \leq \Gamma(x) \\ \Gamma \cdot \vec{y}_1 : \vec{\tau}_1 \vdash_s P \triangleright \mathbf{?}A \otimes C^{\text{-}x}\end{array}}{\Gamma \vdash_s \overline{x}\mathtt{inl}(\boldsymbol{\nu}\,\vec{y}_1 : \vec{\tau}_1).P \triangleright A \otimes \uparrow x \rightarrow B}
\end{array}$$

Here the subtyping is used non-trivially: in (Bra$^{\downarrow}$), the real level of x in Γ is the same or lower than s, so the level elevates. In (Sel$^{\uparrow}$, the real level of x is the same or higher, so the level may go down, but it is recorded in the conclusion. It is notable that this inference crucially depends on the employment of branching as a syntactic construct: without it, these rules should have the same strict conditions as non-linear prefixes.

The final class of rules show the treatment of **!**-**?** modalities and mutability indices, dealing with recursive inputs and their dual outputs, and are most involved. We first have the variable introduction rule (Var$^{!}$), in which we derive $\Gamma \cdot X : \vec{\alpha} \vdash_s X\langle\vec{x}\rangle \triangleright X\langle\vec{x}\rangle$ when we have both $\vdash \alpha_i \leq \Gamma(x_i)$ and $\mathsf{md}(\alpha_0) = \mathbf{!}$, as well as (for consistency with repetitive invocation) $\mathsf{md}(\alpha_i) \in \{\mathbf{?}, \Downarrow, \Uparrow\}$ $(i \neq 0)$. Here we give no restriction on s since when the introduced variable is later bound, all potential tampering at free names would have been recorded except the subject of this recursion, the latter not being tampering. Below we introduce linear recursion rules, for which there are two pairs, one for unary prefix and another for binary prefix. We show the rules for unary input/output.

$$(\mathsf{In}^{\mathbf{!}})\quad \frac{\begin{array}{l}\vdash (\vec{\tau})^{\mathbf{!}}_{s,\kappa} \leq \Gamma(z_0) \quad \vdash \alpha_i \leq \Gamma(z_i)\\ \begin{cases}\Gamma\{\vec{x}/\vec{z}\}\cdot\vec{y}:\vec{\tau}\cdot X:\vec{\alpha} \vdash_s P \triangleright \overrightarrow{\mathtt{p}y}\otimes\mathbf{?}^{\boldsymbol{\iota}}A\{\vec{x}/\vec{z}\}\otimes X\langle\vec{x}\rangle\,(\kappa=\boldsymbol{\iota})\\ \Gamma\{\vec{x}/\vec{z}\}\cdot\vec{y}:\vec{\tau}\cdot X:\vec{\alpha} \vdash_s P \triangleright \overrightarrow{\mathtt{p}y}\otimes\mathbf{?}A\{\vec{x}/\vec{z}\}\otimes X\langle x\vec{w}\rangle(\kappa=\boldsymbol{\mu})\end{cases}\end{array}}{\Gamma \vdash_s (\mu X(\vec{x}:\vec{\alpha}).x_0(\vec{y}:\vec{\tau}).P)\langle\vec{z}\rangle \triangleright \mathbf{!}z_0 \otimes A}$$

$$(\mathsf{Out}^{\mathbf{?}})\quad (\text{where } C/\vec{y} = \downarrow\!\uparrow B)\quad \frac{\begin{array}{l}\vdash (\vec{\tau})^{\mathbf{?}}_{s',\kappa} \leq \Gamma(x) \quad \mathtt{p} \in \{\mathbf{?}, \mathbf{?}^{\boldsymbol{\iota}}\}\\ \Gamma\cdot\vec{y}:\vec{\tau} \vdash_s P \triangleright \mathbf{?}A\otimes C\otimes\mathtt{p}x\\ \kappa=\boldsymbol{\mu} \Rightarrow (s=s' \wedge \mathtt{p}=\mathbf{?})\end{array}}{\Gamma \vdash s \triangleright \overline{x}(\boldsymbol{\nu}\,\vec{y}:\vec{\tau}).PA\otimes B\otimes\mathtt{p}x}$$

In $(\mathrm{In}^{\mathbf{!}})$, we check that the process is immediately recurring to precisely the same behaviour $(X\langle\vec{x}\rangle)$ if it is innocent, or, if it is not innocent, it recurs to the same subject $(X\langle x_0\vec{w}_j\rangle)$. The process can only do free actions with $\mathbf{?}^{\boldsymbol{\iota}}$-modes in the innocent case in addition to the recurrence (except at $\vec{y}$, which are immediately abstracted), so that the process is stateless in its entire visible actions. In the conclusion, the new subject z_0 is introduced with the mode $\mathbf{!}$. In the dual $(\mathrm{Out}^{\mathbf{?}})$, if the prefix is an innocuous output $(\kappa = \boldsymbol{\iota})$, there is no condition on the level of x (s'), so that the level is *not* counted either in the antecedent or in the conclusion (e.g. even if $s' = \bot$ we can have $s \neq \bot$): we are regarding the action as not affecting, and not being affected by, the environment. However if the action is *not* innocuous $(\kappa = \boldsymbol{\mu})$, it is considered as affecting the environment, so that we record its secrecy level by requiring $s' = s$. Note that, even if it is unary, a $\mathbf{?}$-mode output action may indeed affect the environment simply because such an action may or may not exist: just as a unary non-deterministic input/output induces information flow. The corresponding rules for the branching and selection are defined in the same way, see [16].

3.5 Examples of Typing

(Non-linear) Let $\mathtt{sync}^{\Downarrow}_s \overset{\text{def}}{=} ()^{\Downarrow}_s$. Then $a:\overline{\mathtt{sync}^{\Downarrow}_{s'}} \cdot b:\mathtt{sync}^{\Downarrow}_s \vdash_{s'} \overline{a}.b \triangleright \Updownarrow a\otimes\Updownarrow b$, for $s' \leq s$.

(Truly linear) Let $\mathtt{sync}^{\downarrow}_s \overset{\text{def}}{=} ()^{\downarrow}_s$, and its dual $\mathtt{sync}^{\uparrow}_s \overset{\text{def}}{=} ()^{\uparrow}_s$. Then, for arbitrary s and s', we have $a:\mathtt{sync}^{\uparrow}_s \cdot b:\mathtt{sync}^{\downarrow}_{s'} \vdash_{\top} \overline{a}.b \triangleright \uparrow a \to\downarrow b$.

(Branching) Let $\mathtt{bool}^{\mathbf{!}}_s \overset{\text{def}}{=} ([\,\oplus\,]^{\uparrow}_s)^{\mathbf{!}}_s$ be the type of a boolean constant. Then we have $b:\mathtt{bool}^{\mathbf{!}}_s \vdash_s \mathbf{T}\langle b\rangle \triangleright \mathbf{!}b$. For the conditional $\mathbf{If}\langle b, P_1, P_2\rangle$ introduced in §2, suppose that the two branches P_1 and P_2 can be typed at a security level above that of the boolean constant b; that is, P_i is such that $\Gamma\cdot b:\mathtt{bool}^{\mathbf{?}}_{s'} \vdash_s P_i \triangleright \mathbf{?}A\otimes\mathbf{?}^{\boldsymbol{\iota}}b$, for $s' \leq s$. Then $\Gamma \cdot b:\mathtt{bool}^{\mathbf{?}}_{s'} \vdash_s \mathbf{If}\langle b, P_1, P_2\rangle \triangleright A\otimes\mathbf{?}^{\boldsymbol{\iota}}b$. The innocuousness at b is crucial to show that $(\mathtt{bool}^{\mathbf{!}}_{s'})^{\mathbf{?}}_{\top} \leq (\mathtt{bool}^{\mathbf{!}}_{s'})^{\mathbf{?}}_{s'}$ in rule $\mathsf{Out}^{\mathbf{?}}$.

(Copy-cat) The following agent concisely represents the idea of safe information flow in the present calculus. It also serves as a substitute for free name passing for various purposes, including the imperative variable below.

$$[b^s \leftarrow b'^{s'}] \;=\; b(c:\mathtt{bool}^{\uparrow}_s).(\mathbf{If}\langle b', \overline{c}\,\mathtt{inl}, \overline{c}\,\mathtt{inr}\rangle \mid [b \leftarrow b'])$$

This agent transforms a boolean behaviour from b' to b. If $s' \leq s$, then we have: $b:\mathtt{bool}^{\mathbf{!}}_s, b':\mathtt{bool}^{\mathbf{?}}_{s'} \vdash_s [b \leftarrow b'] \triangleright \mathbf{!}b\otimes\mathbf{?}^{\boldsymbol{\iota}}b'$.

(Imperative variable) We give a representation of an imperative variable, alternative to that presented in §2.

$$\mathbf{Var}\langle x^s b^{s'}\rangle^s = x[(z\,{:}\,(\mathtt{bool}^{\boldsymbol{!}}_s)^{\uparrow}_s).(\overline{z}(\boldsymbol{\nu}\, b'\,{:}\,\mathtt{bool}^{\boldsymbol{?}}_s).[b' \leftarrow b]|\mathbf{Var}\langle xb\rangle)\ \&\ (b'\,{:}\,\mathtt{bool}^{\boldsymbol{?}}_s).\mathbf{Var}\langle xb'\rangle]$$

By the copy-cat, sending a new b' has the same effect as sending b. To type this process, let $\mathtt{var}^{\boldsymbol{!}}_s \stackrel{\text{def}}{=} [(\mathtt{bool}^{\boldsymbol{!}}_s)^{\uparrow}_s \& \mathtt{bool}^{\boldsymbol{?}}_s]^{\boldsymbol{!}}_{s,\boldsymbol{\iota}\&\boldsymbol{\mu}}$. Then $x : \mathtt{var}^{\boldsymbol{!}}_s,\ b : \mathtt{bool}^{\boldsymbol{!}}_{s'} \vdash_s \mathbf{Var}\langle x^s b\rangle \triangleright \boldsymbol{!}x \otimes \boldsymbol{?}^{\boldsymbol{\iota}} b$ for $s' \leq s$. Note b has the level s' but the secrecy index is still s, since at b the output is innocuous.

(Assignment) The following offers the typing of the behaviour representing $x^{\mathrm{H}} := y^{\mathrm{L}}$. Let $\mathtt{var}^{\boldsymbol{?}}_s \stackrel{\text{def}}{=} \overline{\mathtt{var}^{\boldsymbol{!}}_s}$ and $\Gamma = x\,{:}\,\mathtt{var}^{\boldsymbol{?}}_{\mathrm{H}} \cdot y\,{:}\,\mathtt{bool}^{\boldsymbol{!}}_{\mathrm{L}}$. Then

$$\Gamma \vdash_{\mathrm{H}} \overline{y}\mathtt{inl}(z\,{:}\,(\mathtt{bool}^{\boldsymbol{?}}_{\mathrm{L}})^{\downarrow}_{\mathrm{L}}).z(b\,{:}\,\mathtt{bool}^{\boldsymbol{?}}_{\mathrm{H}}).\overline{x}\mathtt{inr}(b'\,{:}\,\mathtt{bool}^{\boldsymbol{!}}_{\mathrm{H}}).[b' \leftarrow b] \triangleright \boldsymbol{?}x \otimes \boldsymbol{?}^{\boldsymbol{\iota}} y.$$

4 Elementary Properties of Typed Processes

This section presents the most basic syntactic properties of typed terms. We also briefly discuss one key behavioural property typed terms enjoy. First, the typing system satisfies the standard properties as weakening, strengthening and substitution closure. We only list two important properties. Below (1) says that every typable term has a *canonical typing*, i.e. whenever P is typable, P has the minimum action type and the highest secrecy index, and (2) means that channel types in Γ represent the constraints on the behaviour of P, rather than that of the outside environment (below $A \leq_{\mathtt{G}} A'$ iff $A = A'_0 \otimes \overrightarrow{\boldsymbol{?}^{\boldsymbol{\iota}} x}$ and $A' = A'_0 \otimes \overrightarrow{\boldsymbol{?}x} \otimes \overrightarrow{\updownarrow y} \otimes \overrightarrow{\Updownarrow w}$ for some A'_0).

Proposition 1. (1) (canonical typing) *If $\Gamma \vdash_s P \triangleright A$, then there exists s_0 and A_0 such that $\Gamma \vdash_{s_0} P \triangleright A_0$, and whenever $\Gamma \vdash_{s_1} P \triangleright A_1$ we have $s_1 \leq s_0$ and $A_0 \leq_{\mathtt{G}} A_1$.*

(2) (subsumption-narrowing) *If $\Gamma \cdot x\,{:}\,\alpha \vdash_s P \triangleright A$ and $\alpha \leq \alpha'$, then $\Gamma \cdot x\,{:}\,\alpha' \vdash_s P \triangleright A$.*
Also if $\Gamma \cdot X\,{:}\,\vec{\alpha} \vdash_s P \triangleright A$ and $\alpha_i \geq \beta_i$ for each i, then $\Gamma \cdot X\,{:}\,\vec{\beta} \vdash_s P \triangleright A$.

A fundamental property of the typing system follows. Below $\twoheadrightarrow$ is the multi-step reduction over preterms, defined just as that over untyped terms.

Theorem 1. (subject reduction) *If $\Gamma \vdash_s P \triangleright A$ and $P \twoheadrightarrow Q$ with $\mathsf{bn}(Q) \cap \mathsf{fn}(\Gamma) = \emptyset$, then $\Gamma \vdash_s Q \triangleright A$.*

The theorem says that whatever internal reduction takes place, its composability with the outside, which is controlled by both Γ and A, does not change; and that, moreover, the process is still secure with a no less secrecy index. For the proof, see [16].

The subject reduction is the basis of various significant behavioural properties for typed processes. Here we discuss only one of them, a non-interference property in typed terms (cf. [1,11,25]). A $\langle \Gamma{\cdot}s{\cdot}A\rangle$*-context* is a typed context whose hole is typed under the triple $\langle \Gamma, s, A\rangle$. Then, with respect to security level s, we can define the *s-sensitive maximum sound typed congruence* (cf. [17,28,36]), denoted $\cong_s$, following the standard construction (see [16] for the full definition). We then obtain:

> (**behavioural non-interference**) Let $C[\cdot]$ be a $\langle \Gamma_0 \cdot s_0 \cdot A_0\rangle$-context.
> If $s \lneq s_0$ and $\Gamma_0 \vdash_{s_0} P_i \triangleright A_0$ $(i = 1, 2)$, then $C[P_1] \cong_s C[P_2]$.

The statement says that the behaviour of the whole at lower levels are never affected by its constituting behaviours which only act at higher levels. The proof uses a secrecy-sensitive version of typed bisimilarity, which is a fundamental element of the present theory and which turns out to be a subcongruence of the above maximum sound equality at each secrecy level. By noting ground constants are representable as constant behaviours, one may say the result extends Abadi's non-interference result for ground values [1] to typed process behaviours.

5 Imperative Information Flow as Typed Process Behaviour

5.1 A Multi-Threaded Imperative Calculus

Smith and Volpano [33] presented a type discipline for a basic multi-threaded imperative calculus in which well-typedness ensures secure information flow. In this section we show how the original system can be embedded in the typed calculus introduced in this paper, with a suggestion for a practically interesting extension of the original type discipline through the analysis of the notion of observables. We start with the syntax of untyped phrases of the original calculus, using $x, y, z, \ldots$ for imperative variables.

$$e \ ::= \ x \mid \mathtt{b} \mid e_1 \ \mathtt{and} \ e_2 \qquad\qquad \mathtt{b} \ ::= \ \mathtt{tt} \mid \mathtt{ff}$$

$$c \ ::= \ x := e \mid c_1; c_2 \mid c_1 \mid c_2 \mid \mathtt{if} \ e \ \mathtt{then} \ c_1 \ \mathtt{else} \ c_2 \mid \mathtt{while} \ e \ \mathtt{do} \ c \mid \mathtt{skip}$$

For simplicity we restrict data types to booleans. We also added the `skip` command, and use the parallel composition rather than a system of threads.

The typing system is given in Figure 2. It uses *command types* of form

$$\rho \ ::= \ s \ \mathtt{cmd}_{\Downarrow} \ \mid \ s \ \mathtt{cmd}_{\Uparrow}.$$

Here $s \ \mathtt{cmd}_{\Downarrow}$ (resp. $s \ \mathtt{cmd}_{\Uparrow}$) indicates convergent (resp. divergent) phrases and $s, s', \ldots$ are secrecy levels as before. Note we take secrecy levels from an arbitrary lattice rather than from the two point one. We also use a base E, which is a finite map from variables to secrecy levels. Subsumption in expressions is merged into their typing rules for simplicity. Notice the contravariance in the first two subtyping rules [33,35] and the invariance in the last rule. The types in the original system are embedded into the command types above by setting:

$$(\textsc{h})^\circ \stackrel{\text{def}}{=} \top \qquad (\textsc{l})^\circ \stackrel{\text{def}}{=} \bot \qquad (\textsc{h} \ \mathtt{cmd})^\circ \stackrel{\text{def}}{=} \top \ \mathtt{cmd}_{\Downarrow} \qquad (\textsc{l} \ \mathtt{cmd})^\circ \stackrel{\text{def}}{=} \bot \ \mathtt{cmd}_{\Uparrow},$$

(Subtyping)
$$\frac{s' \le s}{s\ \mathtt{cmd}{\Downarrow} \le s'\ \mathtt{cmd}{\Downarrow}} \qquad \frac{s' \le s}{s\ \mathtt{cmd}{\Downarrow} \le s'\ \mathtt{cmd}{\Uparrow}} \qquad s\ \mathtt{cmd}{\Uparrow} \le s\ \mathtt{cmd}{\Uparrow}$$

(Typing)
$$\textsf{(var)}\ \frac{E(x) \le s}{E \vdash x : s} \qquad \textsf{(bool)}\ E \vdash b : s \qquad \textsf{(and)}\ \frac{E \vdash e_i : s \quad (\mathrm{i} = 1{,}2)}{\vdash e_1\ \mathtt{and}\ e_2 : s}$$

$$\begin{array}{c}\textsf{(skip)}\\ E \vdash \mathtt{skip} : s\ \mathtt{cmd}{\Downarrow}\end{array} \qquad \begin{array}{c}\textsf{(subs)}\\ \dfrac{E \vdash c : \rho \quad \rho \le \rho'}{E \vdash c : \rho'}\end{array} \qquad \begin{array}{c}\textsf{(compose)}\\ \dfrac{E \vdash c_i : \rho}{E \vdash c_1; c_2 : \rho}\end{array} \qquad \begin{array}{c}\textsf{(parallel)}\\ \dfrac{E \vdash c_i : \rho}{E \vdash c_1 \mid c_2 : \rho}\end{array}$$

$$\begin{array}{c}\textsf{(assign)}\\ \dfrac{E \vdash e : s \quad E(x) = s}{E \vdash x := e : s\ \mathtt{cmd}{\Downarrow}}\end{array} \qquad \begin{array}{c}\textsf{(if)}\\ \dfrac{E \vdash e : \mathsf{sec}(\rho) \qquad E \vdash c_i : \rho}{E \vdash \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 : \rho}\end{array} \qquad \begin{array}{c}\textsf{(while)}\\ \dfrac{E \vdash e : \bot \quad E \vdash c : \bot\ \mathtt{cmd}{\Uparrow}}{E \vdash \mathtt{while}\ e\ \mathtt{do}\ c : \bot\ \mathtt{cmd}{\Uparrow}}\end{array}$$

Fig. 2. Typing System of Smith-Volpano calculus

which makes explicit the notion of termination in the original types. With this mapping, the present system is a conservative extension of the original one in both subtyping judgement and typability.

5.2 Embedding

We start with the embedding of types and bases, given in Figure 3. Both command types and bases are translated into two forms, one using channel types and the other using action types. In $[\![\rho]\!]$, a terminating type becomes a truly linear synchronisation type, and a non-terminating type becomes a non-linear synchronisation type, both described in §3.5. $\langle\!\langle \rho \rangle\!\rangle_f$ gives an action type accordingly. We note: given command types ρ, ρ', $\rho \le \rho'$ iff either (1) $\mathsf{sec}([\![\rho]\!]) \ge \mathsf{sec}([\![\rho']\!])$ and both are truly linear unary, (2) $\mathsf{sec}([\![\rho]\!]) \ge \mathsf{sec}([\![\rho']\!])$, $[\![\rho]\!]$ is truly linear unary and $[\![\rho']\!]$ is nonlinear, or (3) $[\![\rho]\!] = [\![\rho']\!]$ and both are nonlinear. This dissects command types into (a) the secrecy level of the whole behaviour (which guarantees the lowest tampering level and which can be degraded by the degradation rule) and (b) the nature of the termination behaviour (noting "non-linear" means a termination action is not guaranteed).

We next turn to the embedding of terms into processes in Figure 3. The framework assumes two boolean constant agents whose behaviours are given in §2.2 and which are shared by all processes, with principal channels $t\!t$ and $f\!\!f$. These free channels are given the ⊥-level, which is in accordance with Smith and Volpano's idea that constants have no secrecy. Following the translation of types, each command becomes a process that upon termination emits an output signal at a channel given as a parameter, typically f (cf. [26]). We are using copy-cat in §3.5 to represent the functionality of value passing. The encoding of terms should be easily understandable, following the known treatment as in [26]: the

(Type and Base)

$$\llbracket s\ \mathtt{cmd}{\Downarrow} \rrbracket \overset{\mathrm{def}}{=} \mathtt{sync}^{\uparrow}_{s} \quad \langle\!\langle s\ \mathtt{cmd}{\Downarrow} \rangle\!\rangle_f \overset{\mathrm{def}}{=} {\uparrow} f \qquad \llbracket \emptyset \rrbracket \overset{\mathrm{def}}{=} \mathit{tt}, \mathit{ff} : \mathtt{var}_{\perp} \qquad \langle\!\langle \emptyset \rangle\!\rangle \overset{\mathrm{def}}{=} \mathbf{?}^{\iota}\mathit{tt} \otimes \mathbf{?}^{\iota}\mathit{ff}$$

$$\llbracket s\ \mathtt{cmd}{\Uparrow} \rrbracket \overset{\mathrm{def}}{=} \mathtt{sync}^{\Uparrow}_{s} \quad \langle\!\langle s\ \mathtt{cmd}{\Uparrow} \rangle\!\rangle_f \overset{\mathrm{def}}{=} {\Updownarrow} f \quad \llbracket E \cdot x : s \rrbracket \overset{\mathrm{def}}{=} \llbracket E \rrbracket \cdot x : \mathtt{var}_s \quad \langle\!\langle E \cdot x : s \rangle\!\rangle \overset{\mathrm{def}}{=} \llbracket E \rrbracket \cdot \mathbf{!}x$$

(Command) $(s = \mathsf{sec}(e)_E$ in all cases, $\mathtt{var}_s \overset{\mathrm{def}}{=} \langle \mathtt{var}^{\mathbf{!}}_{s}, \mathtt{var}^{\mathbf{?}}_{s} \rangle)$

$$\llbracket E \vdash \mathtt{skip} : \rho \rrbracket_f \overset{\mathrm{def}}{=} \overline{f}$$

$$\llbracket E \vdash c_1 ; c_2 : \rho \rrbracket_f \overset{\mathrm{def}}{=} (\boldsymbol{\nu}\, g : \langle \overline{\llbracket \rho \rrbracket}, \llbracket \rho \rrbracket \rangle)(\llbracket E \vdash c_1 : \rho \rrbracket_g \mid g.\llbracket E \vdash c_2 : \rho \rrbracket_f)$$

$$\llbracket E \vdash c_1 \mid c_2 : \rho \rrbracket_f \overset{\mathrm{def}}{=} (\boldsymbol{\nu}\, f_1, f_2 : \langle \overline{\llbracket \rho \rrbracket}, \llbracket \rho \rrbracket \rangle)(\llbracket E \vdash c_1 : \rho \rrbracket_{f_1} \mid \llbracket E \vdash c_2 : \rho \rrbracket_{f_2} \mid f_1.f_2.\overline{f})$$

$$\llbracket E \vdash x := e : \rho \rrbracket_f \overset{\mathrm{def}}{=} \mathbf{eval}\llbracket e \rrbracket^{E}(b^{s'}).\overline{x}\mathtt{inr}(b' : \mathtt{bool}^{\mathbf{!}}_{s'}).([b' \leftarrow b] \mid P) \qquad (s' = E(x))$$

$$\llbracket E \vdash \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 : \rho \rrbracket_f \overset{\mathrm{def}}{=} \mathbf{eval}\llbracket e \rrbracket^{E}(b^{s}).\mathbf{If}\langle b, \llbracket E \vdash c_1 : \rho \rrbracket_f, \llbracket E \vdash c_2 : \rho \rrbracket_f \rangle$$

$$\llbracket E \vdash \mathtt{while}\ e\ \mathtt{do}\ c : \rho \rrbracket_f \overset{\mathrm{def}}{=} (\boldsymbol{\nu} g : \langle \overline{\llbracket \rho \rrbracket}, \llbracket \rho \rrbracket \rangle)(\overline{g} \mid \mathcal{E}\langle f g \vec{x} \rangle) \qquad (E = \{\vec{x} : \vec{s}\}, \alpha_i = \mathtt{var}_{s_i})$$

$$\text{where } \mathcal{E} \overset{\mathrm{def}}{=} \mu X(f, g : \langle \overline{\llbracket \rho \rrbracket}, \llbracket \rho \rrbracket \rangle, \vec{x} : \vec{\alpha}).\ g.\mathbf{eval}\llbracket e \rrbracket^{E}(b^{s}).\mathbf{If}\langle b, (\llbracket E \vdash c : \rho \rrbracket_g | X\langle f g \vec{x} \rangle), \overline{f} \rangle$$

(Expression)

$$\mathbf{eval}\llbracket x \rrbracket^{E}(b^{s}).P \overset{\mathrm{def}}{=} \overline{x}\mathtt{inl}(z : (\mathtt{bool}^{\mathbf{?}}_{s'})^{\downarrow}_{s'}).z(b : \mathtt{bool}^{\mathbf{?}}_{s}).P \qquad (s' = E(x))$$

$$\mathbf{eval}\llbracket \mathtt{tt} \rrbracket^{E}(b^{s}).P \overset{\mathrm{def}}{=} \mathbf{Link}\langle b^{s}, \llbracket \mathtt{b} \rrbracket^{\perp}, P \rangle \qquad (\llbracket \mathtt{tt} \rrbracket \overset{\mathrm{def}}{=} \mathit{tt}, \quad \llbracket \mathtt{ff} \rrbracket \overset{\mathrm{def}}{=} \mathit{ff})$$

$$\mathbf{eval}\llbracket e_1\ \mathsf{and}\ e_2 \rrbracket^{E}(b^{s}).P \overset{\mathrm{def}}{=} \mathbf{eval}\llbracket e_1 \rrbracket^{E}(b_1^{s_1}).\mathbf{eval}\llbracket e_2 \rrbracket^{E}(b_2^{s_2}).$$
$$\mathbf{If}\langle b_1^{s_1}, \mathbf{Link}\langle b^{s}, b_2^{s_2}, P \rangle, \mathbf{Link}\langle b^{s}, b_1^{s_1}, P \rangle \rangle \qquad (s_i = \mathsf{sec}(e_i)_E, s \geq s_1 \sqcup s_2)$$

$$\mathbf{Link}\langle b^{s}, {b'}^{s'}, P \rangle \overset{\mathrm{def}}{=} (\boldsymbol{\nu}\, b : \mathtt{var}_s)(P \mid [b \leftarrow b']) \qquad (s' \leq s)$$

(Security of an expression)

$$\mathsf{sec}(x)_E \overset{\mathrm{def}}{=} E(x) \qquad \mathsf{sec}(\mathtt{b})_E \overset{\mathrm{def}}{=} \perp \qquad \mathsf{sec}(e_1\ \mathsf{and}\ e_2)_E \overset{\mathrm{def}}{=} \mathsf{sec}(e_1)_E \sqcup \mathsf{sec}(e_2)_E$$

Fig. 3. Translation of the Smith-Volpano calculus

interest however lies in how *typability* is transformed via the embedding, and how this transformation sheds light on safe information-flow in the original system. The following theorem underpins this point. Below $\overline{A}$ dualises each mode in A which is assigned to a name, taking $\mathbf{?}$ as the dual of $\mathbf{!}$.

Theorem 2 (Soundness). *If* $E \vdash c : \rho$*, then* $\llbracket E \rrbracket \cdot f : \llbracket \rho \rrbracket \vdash_s \llbracket E \vdash c : \rho \rrbracket_f \triangleright \overline{\langle\!\langle E \rangle\!\rangle} \otimes \langle\!\langle \rho \rangle\!\rangle_f$ *with* $s = \mathsf{sec}(\rho)$*.*

A significant consequence of Theorem 2 is that we obtain, via the non-interference of typed processes mentioned in Section 4, the original non-interference result by Volpano and Smith [33]. The result holds for all terms typable in rules with Figure 2, including typed terms not coming from [33]. As another significant point, the encoding illustrates the reason why the divergent command types cannot be elevated as the convergent ones. Let $E \vdash \mathtt{while}\ e\ \mathtt{do}\ c : s\ \mathtt{cmd}{\Uparrow}$. In the encoding, the body of the loop, which is at level s, depends on the branching at level $\mathsf{sec}(e)_E \leq s$: lowering s can make this dependency dangerous, hence we cannot degrade $s\,\mathtt{cmd}_{\Uparrow}$ as in the convergent types. Also note this argument does not use the restriction $s = \bot$ in the original type discipline.

5.3 Termination as Observable

After the preceding development, a natural question is whether we obtain any new information by doing such an endeavour or not. In this section we outline a technical development which may answer affirmatively to the question.

We first return to the restriction of the original system that allows only the level $\bot$ for divergent commands. This does seem a strong constraint, especially with multiple security levels. How does this constraint appear in process representation? It means we only assign $\mathtt{sync}^{\Uparrow}_{\bot}$ to a channel for the termination, which makes explicit the notion of termination as an observable, both as types and as behaviours. Once we have this notion, we ask what is the real content of having the observable only at $\bot$. Clearly the answer is: "we allow *everybody* to observe the termination." We may then ask what would be the outcome of *not* allowing everybody to observe the termination. Can this make sense? It seems it does: since the time of Multics and as was recently introduced in a widely known programming language [12], a mechanism by which we can prevent processes from even realising the presence of other processes, depending on assigned security levels, is a well-established idea in security, both from integrity and secrecy concerns.

Further, there is a technically important observation that the encoding in Figure 3 does *not* apparently impose restriction on levels of divergent types: indeed the argument for Theorem 2 hardly depends on it. Thus we generalise the while rule as follows.

$$(\mathsf{while})\quad \frac{E \vdash e : s \qquad E \vdash c : s\,\mathtt{cmd}_{\Uparrow}}{E \vdash \mathtt{while}\ e\ \mathtt{do}\ c : s\,\mathtt{cmd}_{\Uparrow}}$$

The new rule is significant in its loosened condition on the guard of the loop, allowing us to type, say, (with M being a level between H and L), $\mathtt{while}\ e^{\mathrm{M}}\ \mathtt{do}\ c\ :\ \mathrm{H}\,\mathtt{cmd}_{\Uparrow}$. With exactly the same encoding, we obtain the soundness result for the extended system with a statement identical to Theorem 2.

Further, this new soundness result leads to a non-interference for the extended imperative calculus just as in the original calculus. The formulation is however different since termination behaviours can change between two initial configurations if we set different values at levels lower than the termination observable. Fixing a base E, let s' be a stipulated level of observability of the termination, and assume there are two environments (assignments of truth values to variables) which are *equivalent with respect to* s', i.e. they only differ in variables at levels higher than s'. Suppose also s is the level of the command type of a well-typed c under E and $s \leq s'$ (thus if c includes a while command, its guard is not affected by the content of variables at levels above s'). Then if c terminates under one of these environments, it will also terminate under the other environment, and the two resulting environments are equivalent with respect to s' (hence with respect to s). If we are without the condition $s \leq s'$, we cannot guarantee the same consequence, though observables except the termination at each state are equivalent with respect to s', related in a coinductive way. See [16] for details. Thus we are again guaranteed secure information flow with added typability, by starting from a typed process representation of imperative program behaviour.

References

1. Abadi, M., Secrecy by typing in security protocols. *TACS'97*, LNCS, 611-637, Springer, 1997.
2. Abadi, M., Banerjee, A., Heintze, N. and Riecke, J., A core calculus of dependency, *POPL'99*, ACM, 1999.
3. Abadi, M., Fournet, C. and Gonthier, G., Secure Communications Processing for Distributed Languages. *LICS'98*, 868–883, IEEE, 1998.
4. Abramsky, S., Computational Interpretation of Linear Logic. *TCS*, vol 111, 1993.
5. Bodei, C, Degano, P., Nielson, F., and Nielson, H. Control Flow Analysis of π-Calculus. *CONCUR'98*, LNCS 1466, 84–98, Springer, 1998.
6. Boudol, G., The pi-calculus in direct style, *POPL'97*, 228–241, ACM, 1997.
7. Denning, D. and Denning, P., Certification of programs for secure information flow. *Communication of ACM*, ACM, 20:504–513, 1997.
8. Focardi, R. and Gorrieri, R., The compositional security checker: A tool for the verification of information flow security properties. *IEEE Transactions on Software Engineering,* 23(9), 1997.
9. Gay, S. and Hole, M., Types and Subtypes for Client-Server Interactions, *ESOP'99*, LNCS 1576, 74–90, Springer, 1999.
10. Girard, J.-Y., Linear Logic, *TCS*, Vol. 50, 1–102, 1987.
11. Goguen, J. and Meseguer, J., Security policies and security models. *IEEE Symposium on Security and Privacy*, 11–20, IEEE, 1982.
12. Gong, L., Prafullchandra, H. and Shcemers, R., Going beyond sandbox: an overview of the new security architecture in the Java development kit 1.2. USENIX Symposium on Internet Technologies and Systems, 1997.
13. Gray, J., Probabilistic interference. *Symposium on Security and Privacy*, 170–179, IEEE, 1990.
14. Honda, K., Types for Dyadic Interaction. *CONCUR'93*, LNCS 715, 509-523, 1993.
15. Honda, K., Composing Processes, *POPL'96*, 344-357, ACM, 1996.
16. A full version of the present paper, QMW CS technical report 767, 1999. Available at `http://www.dcs.qmw.ac.uk/~kohei`.
17. Honda, K. and Yoshida, N. On Reduction-Based Process Semantics. *TCS*. 151, 437-486, 1995.
18. Honda, K. and Yoshida, N. Game-theoretic analysis of call-by-value computation. *TCS*, 221 (1999), 393–456, 1999.
19. Hyland, M. and Ong, L., Pi-calculus, dialogue games and PCF, *FPCA'93*, ACM, 1995.
20. Kobayashi, N., Pierce, B., and Turner, D., Linear types and π-calculus, *POPL'96*, 358–371, 1996.
21. Lafont, Y., Interaction Nets, *POPL'90*, 95–108, ACM, 1990.

22. Larsen, K. and Skou, A. Bisimulation through probabilistic testing. *Information and Computation*, 94:1–28, 1991.
23. Lopes, L, Silva, F. and Vasconcelos, F. A Virtual Machine for the TyCO Process Calculus. *PPDP'99*, 244–260, LNCS 1702, Springer, 1999.
24. Lowe, G. Defining Information Flow. MCS technical report, University of Leicester, 1999/3, 1999.
25. McLean, J. Security models and information flow. *IEEE Symposium on Security and Privacy*, 1990.
26. Milner, R., *Communication and Concurrency*, Prentice-Hall, 1989.
27. Milner, R., Parrow, J.G. and Walker, D.J., A Calculus of Mobile Processes, *Info. & Comp.* 100(1), pp.1–77, 1992.
28. Pierce, B and Sangiorgi.D, Typing and subtyping for mobile processes, *MSCS* 6(5):409–453, 1996.
29. Roscoe, A. W. Intensional Specifications of Security Protocols, *CSFW'96*, IEEE, 1996.
30. Sangiorgi, D. π-calculus, internal mobility, and agent-passing calculi. *TCS*, 167(2):235–271, 1996.
31. Schneider, S. Security properties and CSP. *Symposium on Security and Privacy*, 174–187, 1996.
32. Vasconcelos, V., Typed concurrent objects. *ECOOP'94*, LNCS 821, pp.100–117. Springer, 1994.
33. Volpano, D. and Smith, G., *Secure information flow in a multi-threaded imperative language*, pp.355–364, *POPL'98*, ACM, 1998.
34. Volpano, D. and Smith, G., Language Issues in Mobile Program Security. to appear in LNCS, Springer, 1999.
35. Volpano, D., Smith, G. and Irvine, C., *A Sound type system for secure flow analysis.* J. Computer Security, 4(2,3):167–187, 1996.
36. Yoshida, N. Graph Types for Mobile Processes. *FST/TCS'16*, LNCS 1180, pp.371–386, Springer, 1996. The full version as LFCS Technical Report, ECS-LFCS-96-350, 1996.

Implementing Groundness Analysis with Definite Boolean Functions

Jacob M. Howe and Andy King

Computing Laboratory, University of Kent, CT2 7NF, UK
{j.m.howe, a.m.king}@ukc.ac.uk

Abstract. The domain of definite Boolean functions, *Def*, can be used to express the groundness of, and trace grounding dependencies between, program variables in (constraint) logic programs. In this paper, previously unexploited computational properties of *Def* are utilised to develop an efficient and succinct groundness analyser that can be coded in Prolog. In particular, entailment checking is used to prevent unnecessary least upper bound calculations. It is also demonstrated that join can be defined in terms of other operations, thereby eliminating code and removing the need for preprocessing formulae to a normal form. This saves space and time. Furthermore, the join can be adapted to straightforwardly implement the downward closure operator that arises in set sharing analyses. Experimental results indicate that the new *Def* implementation gives favourable results in comparison with BDD-based groundness analyses.

Keywords: Abstract interpretation, (constraint) logic programs, definite Boolean functions, groundness analysis.

1 Introduction

Groundness analysis is an important theme of logic programming and abstract interpretation. Groundness analyses identify those program variables bound to terms that contain no variables (ground terms). Groundness information is typically inferred by tracking dependencies among program variables. These dependencies are commonly expressed as Boolean functions. For example, the function $x \wedge (y \leftarrow z)$ describes a state in which x is definitely ground, and there exists a grounding dependency such that whenever z becomes ground then so does y.

Groundness analyses usually track dependencies using either *Pos* [3,4,8,15,21], the class of positive Boolean functions, or *Def* [1,16,18], the class of definite positive functions. *Pos* is more expressive than *Def*, but *Def* analysers can be faster [1] and, in practise, the loss of precision for goal-dependent groundness analysis is usually small [18]. This paper is a sequel to [18] and is an exploration of using Prolog as a medium for implementing a *Def* analyser. The rationale for this work was partly to simplify compiler integration and partly to deliver an analyser that was small and thus easy to maintain. Furthermore, it has been suggested that the Prolog user community is not large enough to warrant a compiler vendor to

G. Smolka (Ed.): ESOP/ETAPS 2000, LNCS 1782, pp. 200–214, 2000.

making a large investment in developing an analyser. Thus any analysis that can be quickly prototyped in Prolog is particularly attractive. The main drawback of this approach has traditionally been performance.

The efficiency of groundness analysis depends critically on the way dependencies are represented. C and Prolog based *Def* analysers have been constructed around two representations: (1) Armstrong *et al* [1] argue that Dual Blake Canonical Form (DBCF) is suitable for representing *Def*. This represents functions as conjunctions of definite (propositional) clauses [12] maintained in a normal (orthogonal) form that makes explicit transitive variable dependencies. For example, the function $(x \leftarrow y) \wedge (y \leftarrow z)$ is represented as $(x \leftarrow (y \vee z)) \wedge (y \leftarrow z)$. García de la Banda *et al* [16] adopt a similar representation. It simplifies join and projection at the cost of computing and representing the (extra) transitive dependencies. Introducing redundant dependencies is best avoided since program clauses can (and sometimes do) contain large numbers of variables; the speed of analysis is often related to its memory usage. (2) King *et al* show how meet, join and projection can be implemented with quadratic operations based on a *Sharing* quotient [18]. *Def* functions are essentially represented as a set of models and widening is thus required to keep the size of the representation manageable. Widening trades precision for time and space. Ideally, however, it would be better to avoid widening by, say, using a more compact representation.

This paper contributes to *Def* analysis by pointing out that *Def* has important (previously unexploited) computational properties that enable *Def* to be implemented efficiently and coded straightforwardly in Prolog. Specifically, the paper details:

- how functions can be represented succinctly with non-ground formulae.
- how to compute the join of two formulae without preprocessing the formulae into orthogonal form [1].
- how entailment checking and Prolog machinery, such as difference lists and delay declarations, can be used to obtain a *Def* analysis in which the most frequently used domain operations are very lightweight.
- that the speed of an analysis based on non-ground formulae can compare well against BDD-based *Def* and *Pos* analyses whose domain operations are coded in C [1]. In addition, even without widening, a non-ground formulae analyser can be significantly faster than a *Sharing*-based *Def* analyser [18].

Finally, a useful spin-off of our work is a result that shows how the downward closure operator that arises in BDD-based set sharing analysis [10] can be implemented straightforwardly with standard BDD operations. This saves the implementor the task of coding another BDD operation in C.

The rest of the paper is structured as follows: Section 2 details the necessary preliminaries. Section 3 explains how join can be calculated without resorting to a normal form and also details an algorithm for computing downward closure. Section 4 investigates the frequency of various *Def* operations and explains how representing functions as (non-ground) formulae enables the frequently occurring *Def* operations to be implemented particularly efficiently using, for example, entailment checking. Section 5 evaluates a non-ground *Def* analyser against two BDD analysers. Sections 6 and 7 describe the related and future work, and section 8 concludes.

2 Preliminaries

A Boolean function is a function $f : Bool^n \to Bool$ where $n \geq 0$. A Boolean function can be represented by a propositional formula over X where $|X| = n$. The set of propositional formulae over X is denoted by $Bool_X$. Throughout this paper, Boolean functions and propositional formulae are used interchangeably without worrying about the distinction [1]. The convention of identifying a truth assignment with the set of variables M that it maps to *true* is also followed. Specifically, a map $\psi_X(M) : \wp(X) \to Bool_X$ is introduced defined by: $\psi_X(M) = (\wedge M) \wedge (\neg \vee X \backslash M)$. In addition, the formula $\wedge Y$ is often abbreviated as Y.

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

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

The focus of this paper is on the use of sub-classes of $Bool_X$ in tracing groundness dependencies. These sub-classes are defined below:

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

Note that $Def_X \subseteq Pos_X$. One useful representational property of Def_X is that each $f \in Def_X$ can be described as a conjunction of definite (propositional) clauses, that is, $f = \wedge_{i=1}^{n}(y_i \leftarrow Y_i)$ [12].

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

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

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

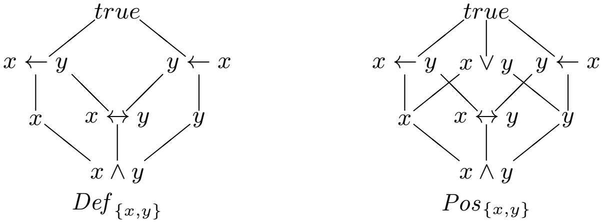


Fig. 1. Hasse diagrams

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

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

The set of (free) variables in a syntactic object o is denoted $\mathit{var}(o)$. Also, $\exists\{y_1, \ldots, y_n\}.f$ (project out) abbreviates $\exists y_1.\ldots.\exists y_n.f$ and $\overline{\exists} Y.f$ (project onto) denotes $\exists \mathit{var}(f) \setminus Y.f$. Let ρ_1, ρ_2 be fixed renamings such that $X \cap \rho_1(X) = X \cap \rho_2(X) = \rho_1(X) \cap \rho_2(X) = \emptyset$. Renamings are bijective and therefore invertible. The downward and upward closure operators $\downarrow$ and $\uparrow$ are defined by $\downarrow f = \mathit{model}_X^{-1}(\{\cap S \mid \emptyset \subset S \subseteq \mathit{model}_X(f)\})$ and $\uparrow f = \mathit{model}_X^{-1}(\{\cup S \mid \emptyset \subset S \subseteq \mathit{model}_X(f)\})$ respectively. Note that $\downarrow f$ has the useful computational property that $\downarrow f = \wedge\{f' \in \mathit{Def}_X \mid f \models f'\}$ if $f \in \mathit{Pos}_X$. Finally, for any $f \in \mathit{Bool}_X$, $\mathit{coneg}(f) = \mathit{model}_X^{-1}(\{X \setminus M \mid M \in \mathit{model}_X(f)\})$.

Example 4. Note that $\mathit{coneg}(x \vee y) = \mathit{model}_{\{x,y\}}^{-1}(\{\{x\}, \{y\}, \emptyset\})$ and therefore $\uparrow \mathit{coneg}(x \vee y) = \mathit{true}$. Hence $\mathit{coneg}(\uparrow \mathit{coneg}(x \vee y)) = \mathit{true} = \downarrow x \vee y$.

This is no coincidence as $\mathit{coneg}(\uparrow \mathit{coneg}(f)) = \downarrow f$. Therefore *coneg* and $\uparrow$ can be used to calculate $\downarrow$.

3 Join and Downward Closure

Calculating join in *Def* is not as straightforward as one would initially think, because of the problem of transitive dependencies. Suppose $f_1, f_2 \in \mathit{Def}_X$ so that $f_i = \wedge F_i$ where $F_i = \{y_1^i \leftarrow Y_1^i, \ldots, y_{n_i}^i \leftarrow Y_{n_i}^i\}$. One naive tactic to compute $f_1 \dot{\vee} f_2$ might be $F = \{y \leftarrow Y_j^1 \wedge Y_k^2 \mid y \leftarrow Y_j^1 \in F_1 \wedge y \leftarrow Y_k^2 \in F_2\}$. Unfortunately, in general, $\wedge F \not\models f_1 \dot{\vee} f_2$ as is illustrated in the following example.

Example 5. Put $F_1 = \{x \leftarrow u, u \leftarrow y\}$ and $F_2 = \{x \leftarrow v, v \leftarrow y\}$ so that $F = \{x \leftarrow u \wedge v\}$, but $f_1 \dot{\vee} f_2 = (x \leftarrow (u \wedge v)) \wedge (x \leftarrow y) \neq \wedge F$. Note, however, that if $F_1 = \{x \leftarrow u, u \leftarrow y, x \leftarrow y\}$ and $F_2 = \{x \leftarrow v, v \leftarrow y, x \leftarrow y\}$ then $F = \{x \leftarrow (u \wedge v), x \leftarrow (u \wedge y), x \leftarrow (v \wedge y), x \leftarrow y\}$ so that $f_1 \dot{\vee} f_2 = \wedge F$.

The problem is that F_i must be explicit about transitive dependencies (this idea is captured in the orthogonal form requirement of [1]). This, however, leads to redundancy in the formula which ideally should be avoided. (Formulae which not necessarily orthogonal will henceforth be referred to as non-orthogonal formulae.)

It is insightful to consider $\dot{\vee}$ as an operation on the models of f_1 and f_2. Since both $model_X(f_i)$ are closed under intersection, $\dot{\vee}$ essentially needs to extend $model_X(f_1) \cup model_X(f_2)$ with new models $M_1 \cap M_2$ where $M_i \in model_X(f_i)$ to compute $f_1 \dot{\vee} f_2$. The following definition expresses this observation and leads to a new way of computing $\dot{\vee}$ in terms of meet, renaming and projection, that does not require formulae to be first put into orthogonal form.

Definition 3. The map $\dot{\curlyvee} : {Bool_X}^2 \to Bool_X$ is defined by: $f_1 \dot{\curlyvee} f_2 = \overline{\exists} Y . f_1 \curlyvee f_2$ where $Y = var(f_1) \cup var(f_2)$ and $f_1 \curlyvee f_2 = \rho_1(f_1) \wedge \rho_2(f_2) \wedge \bigwedge_{y \in Y} y \leftrightarrow (\rho_1(y) \wedge \rho_2(y))$.

Note that $\dot{\curlyvee}$ operates on $Bool_X$ rather than Def_X. This is required for the downward closure operator. Lemma 1 expresses a key relationship between $\dot{\curlyvee}$ and the models of f_1 and f_2.

Lemma 1. Let $f_1, f_2 \in Bool_X$. $M \in model_X(f_1 \dot{\curlyvee} f_2)$ if and only if there exists $M_i \in model_X(f_i)$ such that $M = M_1 \cap M_2$.

Proof. Put $X' = X \cup \rho_1(X) \cup \rho_2(X)$.

Let $M \in model_X(f_1 \dot{\curlyvee} f_2)$. There exists $M \subseteq M' \subseteq X'$ such that $M' \in model_{X'}(f_1 \curlyvee f_2)$. Let $M_i = \rho_i^{-1}(M' \cap \rho_i(Y))$. Observe that $M \subseteq M_1 \cap M_2$ since $(\rho_1(y) \wedge \rho_2(y)) \leftarrow y$. Also observe that $M_1 \cap M_2 \subseteq M$ since $y \leftarrow (\rho_1(y) \wedge \rho_2(y))$. Thus $M_i \in model_X(f_i)$ and $M = M_1 \cap M_2$, as required.

Let $M_i \in model_X(f_i)$ and put $M = M_1 \cap M_2$ and $M' = M \cup \rho_1(M_1) \cup \rho_1(M_2)$. Observe $M' \in model_{X'}(f_1 \curlyvee f_2)$ so that $M \in model_X(f_1 \dot{\curlyvee} f_2)$. ■

From lemma 1 flows the following corollary and also the useful result that $\dot{\curlyvee}$ is monotonic.

Corollary 1. Let $f \in Pos_X$. Then $f = f \dot{\curlyvee} f$ if and only if $f \in Def_X$.

Lemma 2. $\dot{\curlyvee}$ is monotonic, that is, $f_1 \dot{\curlyvee} f_2 \models f_1' \dot{\curlyvee} f_2'$ whenever $f_i \models f_i'$.

Proof. Let $M \in model_X(f_1 \dot{\curlyvee} f_2)$. By lemma 1, there exist $M_i \in model_X(f_i)$ such that $M = M_1 \cap M_2$. Since $f_i \models f_i'$, $M_i \in model_X(f_i')$ and hence, by lemma 1, $M \in model_X(f_1' \dot{\curlyvee} f_2')$. ■

The following proposition states that $\dot{\curlyvee}$ coincides with $\dot{\vee}$ on Def_X. This gives a simple algorithm for calculating $\dot{\vee}$ that does not depend on the representation of a formula.

Proposition 1. Let $f_1, f_2 \in Def_X$. Then $f_1 \dot{\curlyvee} f_2 = f_1 \dot{\vee} f_2$.

Proof. Since $X \models f_2$ it follows by monotonicity that $f_1 = f_1 \dot{\curlyvee} X \models f_1 \dot{\curlyvee} f_2$ and similarly $f_2 \models f_1 \dot{\curlyvee} f_2$. Hence $f_1 \dot{\vee} f_2 \models f_1 \dot{\curlyvee} f_2$ by the definition of $\dot{\vee}$.

Now let $M \in model_X(f_1 \dot{\curlyvee} f_2)$. By lemma 1, there exists $M_i \in model_X(f_i)$ such that $M = M_1 \cap M_2 \in model_X(f_1 \dot{\vee} f_2)$. Hence $f_1 \dot{\curlyvee} f_2 \models f_1 \dot{\vee} f_2$. ■

Downward closure is closely related to $\dot{\curlyvee}$ and, in fact, $\dot{\curlyvee}$ can be used repeatedly to compute a finite iterative sequence that converges to $\downarrow$. This is stated in proposition 2. Finiteness follows from bounded chain length of Pos_X.

Proposition 2. Let $f \in Pos_X$. Then $\downarrow f = \vee_{i \geq 1} f_i$ where $f_i \in Pos_X$ is the increasing chain given by: $f_1 = f$ and $f_{i+1} = f_i \dot{\curlyvee} f_i$.

Proof. Let $M \in model_X(\downarrow f)$. Thus there exists $M_j \in model_X(f)$ such that $M = \cup_{j=1}^{m} M_j$. Observe $M_1 \cap M_2, M_3 \cap M_4, \ldots \in model_X(f_2)$ and therefore $M \in model_X(f_{\lceil \log_2(m) \rceil})$. Since $m \leq 2^{2^n}$ where $n = |X|$ it follows that $\downarrow f \models f_{2^n}$.

Proof by induction is used for the opposite direction. Observe that $f_1 \models \downarrow f$. Suppose $f_i \models \downarrow f$. Let $M \in model_X(f_{i+1})$. By lemma 1 there exists $M_1, M_2 \in model_X(f_i)$ such that $M = M_1 \cap M_2$. By the inductive hypothesis $M_1, M_2 \in model_X(\downarrow f)$ thus $M \in model_X(\downarrow f)$. Hence $f_{i+1} \models \downarrow f$.

Finally, $\vee_{i=1} f_i \in Def_X$ since $f_1 \in Pos_X$ and $\dot{\curlyvee}$ is monotonic and thus $X \in model_X(\vee_{i=1} f_i)$. ■

The significance of this is that it enables $\downarrow$ to be computed in terms of existing BDD operations thus freeing the implementor from more low level coding.

4 Design and Implementation

There are typically many degrees of freedom in designing an analyser, even for a given domain. Furthermore, work can often be shifted from one abstract operation into another. For example, García de la Banda *et al* [16] maintain DBCF by a meet that uses six rewrite rules to normalise formulae. This gives a linear time join and projection at the expense of an exponential meet. Conversely, King *et al* [18] have meet, join and projection operations that are quadratic in the number of models. Note, however, that the numbers of models is exponential (explaining the need for widening). Ideally, an analysis should be designed so that the most frequently used operations have low complexity and are therefore fast.

4.1 Frequency Analysis

In order to balance the frequency of an abstract operation against its cost, a BDD-based *Def* analyser was implemented and instrumented to count the number of calls to the various abstract operations. The BDD-based *Def* analyser is coded in Prolog as a simple meta-interpreter that uses induced magic-sets [7] and eager evaluation [22] to perform goal-dependent bottom-up evaluation.

Induced magic is a refinement of the magic set transformation, avoiding much of the re-computation that arises because of the repetition of literals in the bodies of magicked clauses [7]. It also avoids the overhead of applying the magic set transformation. Eager evaluation [22] is a fixpoint iteration strategy which proceeds as follows: whenever an atom is updated with a new (less precise) abstraction, a recursive procedure is invoked to ensure that every clause that has that atom in its body is re-evaluated. Induced magic may not be as efficient as, say, GAIA [19] but it can be coded easily in Prolog.

The BDD-based *Def* analysis is built on a ROBDD package coded by Armstrong and Schachte [1]. The package is intended for *Pos* analysis and therefore supplies a $\vee$ join rather than a $\dot{\vee}$ join. The package did contain, however, a hand-crafted C upward closure operator $\uparrow$ enabling $\dot{\vee}$ to be computed by $f_1 \dot{\vee} f_2 = \downarrow(f_1 \vee f_2)$ where $\downarrow f = \mathit{coneg}(\uparrow \mathit{coneg}(f))$. The operation $\mathit{coneg}(f)$ can be computed simply by interchanging the left and right (true and false) branches of an ROBDD. The analyser also uses the environment trimming tactic used by Schachte to reduce the number of variables that occur in a ROBDD. Specifically, clause variables are numbered and each program point is associated with a number, in such a way that if a variable has a number less than that associated with the program point, then it is redundant (does not occur to the right of the program point) and hence can be projected out. This optimisation is important in achieving practical analysis times for some large programs.

The following table gives a breakdown of the number of calls to each abstract operation in the BDD-based *Def* analysis of eight large programs. Meet, join, equiv, project and rename are the obvious Boolean operations. Join (diff) is the number of calls to a join $f_1 \dot{\vee} f_2$ where $f_1 \dot{\vee} f_2 \neq f_1$ and $f_1 \dot{\vee} f_2 \neq f_2$. Project (trim) are the number of calls to project that stem from environment trimming.

file	strips	chat_parser	sim_v5-2	peval	aircraft	essln	chat_80	aqua_c
meet	815	4471	2192	2198	7063	8406	15483	112455
join	236	1467	536	632	2742	1668	4663	35007
join (diff)	33	243	2	185	26	177	693	5173
equiv	236	1467	536	632	2742	1668	4663	35007
project	330	1774	788	805	3230	2035	5523	38163
project (trim)	173	1384	770	472	2082	2376	5627	42989
rename	857	4737	2052	2149	8963	5738	14540	103795

Observe that meet and rename are called most frequently and therefore, ideally, should be the most lightweight. Project, project (trim), join and equiv calls occur with similar frequency but note that it is rare for a join to differ from both its arguments. Join is always followed by an equivalence and this explains why the join and equiv rows coincide.

Next, the complexity of ROBDD and DBCF (specialised for *Def* [1]) operations are reviewed in relation to their calling frequency. Suggestions are made about balancing the complexity of an operation against its frequency by using a non-orthogonal formulae representation.

For ROBDDs (DBCF) meet is quadratic (exponential) in the size of its arguments [1]. For ROBDDs (DBCF) these arguments are exponential (polynomial) in the number of variables. Representing *Def* functions as non-orthogonal formulae is attractive since meet is concatenation which can be performed in constant time (using difference lists). Renaming is quadratic for ROBDDs (linear for DBCF) in the size of its argument [1]. Renaming a non-orthogonal formula is $O(m \log(n))$ where m (n) is the number of symbols (variables) in its argument.

For ROBDDs (DBCF), join is quadratic (quartic) in the size of its arguments [1]. For non-orthogonal formulae, join is exponential. Note, however, that the majority of joins result in one of the operands and hence are unnecessary. This can be detected by using an entailment check which is quadratic in the size of the representation. Thus it is sensible to filter join through an entailment check so that join is called comparatively rarely. Therefore its complexity is less of an issue. Specifically, if $f_1 \models f_2$ then $f_1 \dot{\vee} f_2 = f_2$. For ROBDDs, equivalence checking is constant time, whereas for DBCF it is linear in the size of the representation. For non-orthogonal formulae, equivalence is quadratic in the size of the representation. Observe that meet occurs more frequently than equality and therefore a gain should be expected from trading an exponential meet and a linear join for a constant time meet and an exponential join.

For ROBDDs (DBCF), projection is quadratic (linear) in the size of its arguments [1]. For a non-orthogonal representation, projection is exponential, but again, entailment checking can be used to prevent the majority of projections.

4.2 The GEP Representation

A call (or answer) pattern is a pair $\langle a, f \rangle$ where a is an atom and $f \in \mathit{Def}_{var(a)}$. Normally the arguments of a are distinct variables. The formula f is a conjunction (list) of propositional Horn clauses in the *Def* analysis described in this paper. In a non-ground representation the arguments of a can be instantiated and aliased to express simple dependency information [9]. For example, if $a = p(x_1, ..., x_5)$, then the atom $p(x_1, \mathit{true}, x_1, x_4, \mathit{true})$ represents a coupled with the formula $(x_1 \leftrightarrow x_3) \wedge x_2 \wedge x_5$. This enables the abstraction $\langle p(x_1, ..., x_5), f_1 \rangle$ to be collapsed to $\langle p(x_1, \mathit{true}, x_1, x_4, \mathit{true}), f_2 \rangle$ where $f_1 = (x_1 \leftrightarrow x_3) \wedge x_2 \wedge x_5 \wedge f_2$. This encoding leads to a more compact representation and is similar to the GER factorisation of ROBDDs proposed by Bagnara and Schachte [3]. The representation of call and answer patterns described above is called GEP (groundness, equivalences and propositional clauses) where the atom captures the first two properties and the formula the latter. Note that the current implementation of the GEP representation does not avoid inefficiencies in the representation such as the repetition of *Def* formulae.

4.3 Abstract Operations

The GEP representation requires the abstract operations to be lifted from Boolean formulae to call and answer patterns.

Meet The meet of the pairs $\langle a_1, f_1 \rangle$ and $\langle a_2, f_2 \rangle$ can be computed by unifying a_1 and a_2 and concatenating f_1 and f_2.

Renaming The objects that require renaming are formulae and call (answer) pattern GEP pairs. If a dynamic database is used to store the pairs [17], then renaming is automatically applied each time a pair is looked-up in the database. Formulae can be renamed with a single call to the Prolog builtin copy_term.

Join Calculating the join of the pairs $\langle a_1, f_1 \rangle$ and $\langle a_2, f_2 \rangle$ is complicated by the way that join interacts with renaming. Specifically, in a non-ground representation, call (answer) patterns would be typically stored in a dynamic database so that $var(a_1) \cap var(a_2) = \emptyset$. Hence $\langle a_1, f_1 \rangle$ (or equivalently $\langle a_2, f_2 \rangle$) have to be appropriately renamed before the join is calculated. This is achieved as follows. Plotkin's anti-unification algorithm [20] is used to compute the most specific atom a that generalises a_1 and a_2. The basic idea is to reformulate a_1 as a pair $\langle a'_1, f'_1 \rangle$ which satisfies two properties: a'_1 is a syntactic variant of a; the pair represents the same dependency information as $\langle a_1, true \rangle$. A pair $\langle a'_2, f'_2 \rangle$ is likewise constructed that is a reformulation of a_2. The atoms a, a'_1 and a'_2 are unified and then the formula $f = (f_1 \wedge f'_1) \dot{\curlyvee} (f_2 \wedge f'_2)$ is calculated as described in section 3 to give the join $\langle a, f \rangle$. In actuality, the computation of $\langle a'_1, f'_1 \rangle$ and the unification $a = a'_1$ can be combined in a single pass as is outlined below. Suppose $a = p(t_1, \ldots, t_n)$ and $a_1 = p(s_1, \ldots, s_n)$. Let $g_0 = true$. For each $1 \leq k \leq n$, one of the following cases is selected. (1) If t_k is syntactically equal to s_k, then put $g_k = g_{k-1}$. (2) If s_k is bound to $true$, then put $g_k = g_{k-1} \wedge (t_k \leftarrow true)$. (3) If $s_k \in var(\langle s_1, \ldots, s_{k-1} \rangle)$, then unify s_k and t_k and put $g_k = g_{k-1}$. (4) Otherwise, put $g_k = g_{k-1} \wedge (t_k \leftarrow s_k) \wedge (s_k \leftarrow t_k)$. Finally, let $f'_1 = g_n$. The algorithm is applied analogously to bind variables in a and construct f'_2. The join of the pairs is then given by $\langle a, (f_1 \wedge f'_1) \dot{\curlyvee} (f_2 \wedge f'_2) \rangle$.

Example 6. Consider the join of the GEP pairs $\langle a_1, true \rangle$ and $\langle a_2, y_1 \leftarrow y_2 \rangle$ where $a_1 = p(true, x_1, x_1, x_1)$ and $a_2 = p(y_1, y_2, true, true)$. The most specific generalisation of a_1 and a_2 is $a = p(z_1, z_2, z_3, z_3)$. The table below illustrates the construction of $\langle a'_1, f'_1 \rangle$ and $\langle a'_2, f'_2 \rangle$ in the left- and right-hand columns.

k	$case$	g_k	θ_k	$case'$	g'_k	θ'_k
0		$true$	ϵ		$true$	ϵ
1	2	$z_1 \leftarrow true$	ϵ	4	$y_1 \leftrightarrow z_1$	ϵ
2	4	$g_1 \wedge (z_2 \leftrightarrow x_1)$	θ_1	4	$g'_1 \wedge (y_2 \leftrightarrow z_2)$	θ_1
3	3	g_2	$\{x_1 \mapsto z_3\}$	2	$g'_2 \wedge (z_3 \leftarrow true)$	θ_1
4	1	g_2	θ_3	2	$g'_3 \wedge (z_3 \leftarrow true)$	θ_1

Putting $\theta = \theta'_4 \circ \theta_4 = \{x_1 \mapsto z_3\}$, the join is given by $\langle \theta(a), \theta(g_4 \wedge true) \dot{\curlyvee} \theta(g'_4 \wedge y_1 \leftarrow y_2) \rangle = \langle a, (z_1 \leftarrow true) \wedge (z_2 \leftrightarrow z_3) \dot{\curlyvee} (y_1 \leftrightarrow z_1) \wedge (y_2 \leftrightarrow z_2) \wedge (z_3 \leftarrow true) \wedge (y_1 \leftarrow y_2) \rangle = \langle p(z_1, z_2, z_3, z_3), (z_1 \leftarrow z_2) \wedge (z_3 \leftarrow z_2) \rangle$.

Note that often a_1 is a variant of a_2. This can be detected with a lightweight variance check, enabling join and renaming to be reduced to unifying a_1 and a_2 and computing $f = f_1 \dot{\curlyvee} f_2$ to give the pair $\langle a_1, f \rangle$.

Projection Projection is only applied to formulae. Each of the variables to be projected out is eliminated in turn, as follows. Suppose x is to be projected out of f. First, all those clauses with x as their head are found, giving $\{x \leftarrow X_i \mid i \in I\}$ where I is a (possibly empty) index set. Second, all those clauses with x in the body are found, giving $\{y \leftarrow Y_j \mid j \in J\}$ where J is a (possibly empty) index set. Thirdly these clauses of f are replaced by $\{y \leftarrow Z_{i,j} \mid i \in I \wedge j \in J \wedge Z_{i,j} = X_i \cup (Y_j \setminus \{x\}) \wedge y \notin Z_{i,j}\}$ (syllogizing). Fourthly, a compact representation is maintained by eliminating redundant clauses (absorption). By appropriately ordering the clauses, all four steps can be performed in a single pass over f. A final pass over f retracts clauses such as $x \leftarrow true$ by binding x to true and also removes clause pairs such as $y \leftarrow z$ and $z \leftarrow y$ by unifying y and z.

Entailment Entailment checking is only applied to formulae. A forward chaining decision procedure for propositional Horn clauses (and hence *Def*) is used to test entailment. A non-ground representation allows chaining to be implemented efficiently using block declarations. To check that $\wedge_{i=1}^{n} y_i \leftarrow Y_i$ entails $z \leftarrow Z$ the variables of Z are first grounded. Next, a process is created for each clause $y_i \leftarrow Y_i$ that blocks until Y_i is ground. When Y_i is ground, the process resumes and grounds y_i. If z is ground after a single pass over the clauses, then $(\wedge_{i=1}^{n} y_i \leftarrow Y_i) \models z \leftarrow Z$. By calling the check under negation, no problematic bindings or suspended processes are created.

5 Experimental Evaluation

A *Def* analyser using the non-ground techniques described in this paper has been implemented. This implementation is built in Prolog using the same induced magic framework as for the BDD-based *Def* analyser, therefore the analysers work in lock step and generate the same results. (The only difference is that the non-ground analyser does not implement environment trimmed since the representation is far less sensitive to the number of variables in a clause.) The core of the analyser (the fixpoint engine) is approximately 400 lines of code and took one working week to write, debug and tune.

In order to investigate whether entailment checking, the join ($\dot{\curlyvee}$) algorithm, and the GEP representation are enough to obtain a fast and scalable analysis, the non-ground analyser was compared with the BDD-based analyser for speed and scalability. Since King *et al* [18] do not give precision results for *Pos* for larger benchmarks, we have also implemented a BDD-based *Pos* analyser in the same vein, so that firmer conclusions about the relative precision of *Def* and *Pos* can be drawn. It is reported in [2], [3] that a hybrid implementation of ROBDDs, separating maintenance of definiteness information and of various forms of dependency information can give significantly improved performance. Therefore, it is to be expected that an analyser based on such an implementation of ROBDDs would be faster than that used here.

The comparisons focus on goal-dependent groundness analysis of 60 Prolog and CLP($\mathcal{R}$) programs. The results are given in the table below. In this table, the size column gives the number of distinct (abstract) clauses in the programs. The abs column gives the time for parsing the files and abstracting them, that is, replacing built-ins, such as arg(x, t, s), with formulae, such as $x \wedge (s \leftarrow t)$.

			fixpoint			precision		
file	size	abs	*Def*$_{NG}$	*Def*$_{BDD}$	*Pos*	*Def*	*Pos*	%
rotate.pl	3	0.00	0.00	0.00	0.00	3	6	50
circuit.clpr	20	0.02	0.02	0.03	0.02	3	3	0
air.clpr	20	0.02	0.02	0.03	0.02	9	9	0
dnf.clpr	23	0.02	0.01	0.01	0.01	8	8	0
dcg.pl	23	0.02	0.01	0.01	0.02	59	59	0
hamiltonian.pl	23	0.02	0.01	0.01	0.01	37	37	0
poly10.pl	29	0.02	0.00	0.00	0.01	0	0	0
semi.pl	31	0.03	0.03	0.28	0.28	28	28	0
life.pl	32	0.02	0.01	0.02	0.02	58	58	0
rings-on-pegs.clpr	34	0.02	0.02	0.04	0.04	11	11	0
meta.pl	35	0.01	0.01	0.02	0.01	1	1	0
browse.pl	36	0.02	0.01	0.02	0.02	41	41	0
gabriel.pl	38	0.02	0.01	0.03	0.03	37	37	0
tsp.pl	38	0.03	0.01	0.04	0.04	122	122	0
nandc.pl	40	0.03	0.01	0.03	0.03	37	37	0
csg.clpr	48	0.04	0.01	0.02	0.02	12	12	0
disj_r.pl	48	0.02	0.01	0.04	0.04	97	97	0
ga.pl	48	0.06	0.01	0.04	0.04	141	141	0
critical.clpr	49	0.03	0.03	0.04	0.04	14	14	0
scc1.pl	51	0.03	0.01	0.06	0.04	89	89	0
mastermind.pl	53	0.04	0.01	0.04	0.04	43	43	0
ime_v2-2-1.pl	53	0.04	0.03	0.09	0.08	101	101	0
robot.pl	53	0.03	0.00	0.01	0.01	41	41	0
cs_r.pl	54	0.05	0.01	0.04	0.04	149	149	0
tictactoe.pl	56	0.06	0.01	0.03	0.04	60	60	0
flatten.pl	56	0.03	0.04	0.09	0.08	27	27	0
dialog.pl	61	0.02	0.01	0.03	0.03	70	70	0
map.pl	66	0.02	0.01	0.08	0.08	17	17	0
neural.pl	67	0.05	0.01	0.05	0.05	123	123	0
bridge.clpr	69	0.08	0.01	0.02	0.03	24	24	0
conman.pl	71	0.04	0.00	0.02	0.02	6	6	0
kalah.pl	78	0.04	0.02	0.04	0.04	199	199	0
unify.pl	79	0.04	0.07	0.12	0.10	70	70	0
nbody.pl	85	0.07	0.06	0.10	0.11	113	113	0
peep.pl	86	0.11	0.03	0.06	0.05	10	10	0
boyer.pl	95	0.06	0.04	0.04	0.05	3	3	0
bryant.pl	95	0.07	0.20	0.15	0.15	99	99	0
sdda.pl	99	0.05	0.06	0.06	0.06	17	17	0
read.pl	105	0.07	0.06	0.11	0.10	99	99	0
press.pl	109	0.07	0.11	0.16	0.18	53	53	0
qplan.pl	109	0.08	0.02	0.08	0.07	216	216	0
trs.pl	111	0.11	0.11	0.31	0.60	13	13	0
reducer.pl	113	0.07	0.11	0.16	0.14	41	41	0
simple_analyzer.pl	140	0.09	0.13	0.34	0.44	89	89	0
dbqas.pl	146	0.09	0.02	0.05	0.05	43	43	0
ann.pl	148	0.09	0.11	0.24	0.23	74	74	0
asm.pl	175	0.14	0.06	0.14	0.13	90	90	0
nand.pl	181	0.12	0.04	0.21	0.19	402	402	0
rubik.pl	219	0.16	0.15	0.39	0.40	158	158	0
lnprolog.pl	221	0.10	0.08	0.14	0.14	143	143	0
ili.pl	225	0.15	0.25	0.23	0.24	4	4	0
sim.pl	249	0.18	0.39	0.56	0.52	100	100	0
strips.pl	261	0.17	0.01	0.11	0.11	142	142	0
chat_parser.pl	281	0.21	0.45	0.59	0.60	505	505	0
sim_v5-2.pl	288	0.17	0.05	0.20	0.20	455	457	0.4
peval.pl	328	0.16	0.28	0.27	0.27	27	27	0
aircraft.pl	397	0.48	0.14	0.55	0.59	687	687	0
essln.pl	565	0.36	0.21	0.58	0.58	163	163	0
chat_80.pl	888	0.92	1.31	1.89	2.27	855	855	0
aqua_c.pl	4009	2.48	11.29	104.99	897.10	1288	1288	0

The abstracter deals with meta-calls, asserts and retracts following the elegant (two program) scheme detailed by Bueno *et al* [6]. The fixpoint columns give the time, in seconds, to compute the fixpoint for each of the three analysers (Def_{NG} and Def_{BDD} denote respectively the non-ground and BDD-based *Def* analyser). The precision columns give the total number of ground arguments in the call and answer patterns (and exclude those ground arguments for predicates introduced by normalising the program into definite clauses). The % column express the loss of precision by *Def* relative to *Pos*. All three analysers were coded in SICStus 3.7 and the experiments performed on a 296MHz Sun UltraSPARC-II with 1GByte of RAM running Solaris 2.6.

The experimental results indicate the precision of *Def* is close to that of *Pos*. Although rotate.pl is small it has been included in the table because it was the only program for which significant precision was lost. Thus, whilst it is always possible to construct programs in which disjunctive dependency information (which cannot be traced in *Def*) needs to be tracked to maintain precision, these results suggest that *Def* is adequate for top-down groundness analysis of many programs.

The speed of the non-ground *Def* analyser compares favourably with both the BDD analysers. This is surprising because the BDD analysers make use of hashing and memoisation to avoid repeated work. In the non-ground *Def* analyser, the repeated work is usually in meet and entailment checking, and these operations are very lightweight. In the larger benchmarks, such as aqua_c.pl, the BDD analysis becomes slow as the BDDs involved are necessarily large. Widening for BDDs can make such examples more manageable [15]. Notice that the time spent in the core analyser (the fixpoint engine) is of the same order as that spent in the abstracter. This suggests that a large speed up in the analysis time needs to be coupled with a commensurate speedup in the abstracter.

To give an initial comparison with the *Sharing*-based *Def* analyser of King *et al* [18], the clock speed of the Sparc-20 used in the *Sharing* experiments has been used to scale the results in this paper. These findings lead to the preliminary conclusion that the analysis presented in this paper is about twice as fast as the *Sharing* quotient analyser. Furthermore, this analyser relies on widening to keep the abstractions small, hence may sacrifice some precision for speed.

6 Related Work

Van Hentenryck *et al* [21] is an early work which laid a foundation for BDD-based *Pos* analysis. Corsini *et al* [11] describe how variants of *Pos* can be implemented using Toupie, a constraint language based on the μ-calculus. If this analyser was extended with, say, magic sets, it might lead to a very respectable goal-dependent analysis. More recently, Bagnara and Schachte [3] have developed the idea [2] that a hybrid implementation of a ROBDD that keeps definite information separate from dependency information is more efficient than keeping the two together. This hybrid representation can significantly decrease the size of an ROBDD and thus is a useful implementation tactic.

Armstrong *et al* [1] study a number of different representations of Boolean function for both *Def* and *Pos*. An empirical evaluation on 15 programs suggests that specialising Dual Blake Canonical Form (DBCF) for *Def* leads to the fastest analysis overall. This representation of a *Def* function f is in orthogonal form since it is constructed from *all* the prime consequents that are entailed by f. It thus includes redundant transitive dependencies. Armstrong *et al* [1] also perform interesting precision experiments. *Def* and *Pos* are compared, however, in a bottom-up framework that is based on condensing which is therefore biased towards *Pos*. The authors point out that a top-down analyser would improve the precision of *Def* relative to *Pos* and our work supports this remark.

García de la Banda *et al* [16] describe a Prolog implementation of *Def* that is also based on an orthogonal DBCF representation (though this is not explicitly stated) and show that it is viable for some medium sized benchmarks. Fecht [15] describes another groundness analyser that is not coded in C. Fecht adopts ML as a coding medium in order to build an analyser that is declarative and easy to maintain. He uses a sophisticated fixpoint solver and his analysis times compare favourably with those of Van Hentenryck *et al* [21].

Codish and Demoen [8] describe a non-ground model based implementation technique for *Pos* that would encode $x_1 \leftrightarrow (x_2 \wedge x_3)$ as three tuples $\langle true, true, true\rangle$, $\langle false, \_, false\rangle$, $\langle false, false, \_\rangle$. Codish *et al* [9] propose a sub-domain of *Def* that can only propagate dependencies of the form $(x_1 \leftrightarrow x_2) \wedge x_3$ across procedure boundaries. The main finding of Codish *et al* [9] is that this sub-domain loses only a small amount of precision for goal-dependent analysis.

King *et al* [18] show how the equivalence checking, meet and join of *Def* can be efficiently computed with a *Sharing* quotient. Widening is required to keep the representation manageable.

Finally, a curious connection exists between the join algorithm described in this paper and a relaxation that occurs in disjunctive constraint solving [14]. The relaxation computes the join (closure of the convex hull) of two polyhedra P_1 and P_2 where $P_i = \{\boldsymbol{x} \in \mathbb{R}^n \mid A_i\boldsymbol{x} \leq B_i\}$. The join of P_1 and P_2 can be expressed as:

$$P = \left\{ \boldsymbol{x} \in \mathbb{R}^n \;\middle|\; \begin{array}{c} A_1\rho_1(\boldsymbol{x}) \leq B_1 \wedge A_2\rho_2(\boldsymbol{x}) \leq B_2 \wedge \\ 0 \leq \lambda \leq 1 \wedge \boldsymbol{x} = \lambda\rho_1(\boldsymbol{x}) + (1-\lambda)\rho_2(\boldsymbol{x}) \end{array} \right\}$$

which amounts to the same tactic of constructing join in terms of meet (conjunction of linear equations), renaming (ρ_1 and ρ_2) and projection (the variables of interest are $\boldsymbol{x}$).

7 Future Work

Initial profiling has suggested that a significant proportion of the analysis time is spent projecting onto (new) call and answer patterns, so recoding this operation might impact on the speed of the analysis. Also, a practical comparison with a DBCF analyser would be insightful. This is the immediate future work. In the medium term, it would be interesting to apply widening to obtain an analysis with polynomial guarantees. Time complexity relates to the maximum number of iterations of a fixpoint analysis and this, in turn, depends on the length of the longest ascending chain in the underlying domain. For both Pos_X and Def_X the longest chains have length $2^n - 1$ where $|X| = n$ [18]. One way to accelerate the analysis, would be to widen call and answer patterns by discarding the formulae component of the GEP representation if the number of updates to a particular call or answer pattern exceeded, say, 8 [18]. The abstraction then corresponds to an $EPos_X$ function whose chain length is linear in X [9]. Although widening for space is not as critical as in [18], this too would be a direction for future work. In the long term, it would be interesting to apply *Def* to other dependency analysis problems, for example, strictness [13] and finiteness [5] analysis.

The frequency analysis which has been used in this paper to tailor the costs of the abstract operations to the frequency with which they are called could be applied to other analyses, such as type, freeness or sharing analyses.

8 Conclusions

The representation and abstract operations for *Def* have been chosen by following a strategy. The strategy was to design an implementation so as to ensure that the most frequently called operations are the most lightweight. Previously unexploited computational properties of *Def* have been used to avoid expensive joins (and projections) through entailment checking; and to keep abstractions small by reformulating join in such a way as to avoid orthogonal reduced monotonic body form. The join algorithm has other applications such as computing the downward closure operator that arises in BDD-based set sharing analysis.

By combining the techniques described in this paper, an analyser has been constructed that is precise, can be implemented easily in Prolog, and whose speed compares favourably with BDD-based analysers.

Acknowledgements We thank Mike Codish, Roy Dyckhoff and Andy Heaton for useful discussions. We would also like to thank Peter Schachte for help with his BDD analyser. This work was funded partly by EPSRC Grant GR/MO8769.

References

1. T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Two Classes of Boolean Functions for Dependency Analysis. *Science of Computer Programming*, 31(1):3–45, 1998.
2. R. Bagnara. A Reactive Implementation of *Pos* using ROBDDs. In *Programming Languages: Implementation, Logics and Programs*, volume 1140 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 1996.
3. R. Bagnara and P. Schachte. Factorizing Equivalent Variable Pairs in ROBDD-Based Implementations of Pos. In *Seventh International Conference on Algebraic Methodology and Software Technology*, volume 1548 of *Lecture Notes in Computer Science*, pages 471–485. Springer, 1999.

4. N. Baker and H. Søndergaard. Definiteness Analysis for CLP($\mathcal{R}$). In *Australian Computer Science Conference*, pages 321–332, 1993.
5. P. Bigot, S. Debray, and K. Marriott. Understanding Finiteness Analysis using Abstract Interpretation. In *Joint International Conference and Symposium on Logic Programming*, pages 735–749. MIT Press, 1992.
6. F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Global Analysis of Standard Prolog Programs. In *European Symposium on Programming*, volume 1058 of *Lecture Notes in Computer Science*, pages 108–124. Springer, 1996.
7. M. Codish. Efficient Goal Directed Bottom-up Evaluation of Logic Programs. *Journal of Logic Programming*, 38(3):355–370, 1999.
8. M. Codish and B. Demoen. Analysing Logic Programs using "prop"-ositional Logic Programs and a Magic Wand. *Journal of Logic Programming*, 25(3):249–274, 1995.
9. M. Codish, A. Heaton, A. King, M. Abo-Zaed, and P. Hill. Widening Positive Boolean Functions for Goal-dependent Groundness Analysis. Technical Report 12-98, Computing Laboratory, May 1998. http://www.cs.ukc.ac.uk/pubs/1998/589.
10. M. Codish, H. Søndergaard, and P. Stuckey. Sharing and Groundness Dependencies in Logic Programs. *ACM Transactions on Programming Languages and Systems*, 1999. To appear.
11. M.-M. Corsini, K. Musumbu, A. Rauzy, and B. Le Charlier. Efficient Bottom-up Abstract Interpretation of Prolog by means of Constraint Solving over Finite Domains. In *Programming Language Implementation and Logic Programming*, volume 714 of *Lecture Notes in Computer Science*, pages 75–91. Springer, 1993.
12. P. Dart. On Derived Dependencies and Connected Databases. *Journal of Logic Programming*, 11(1&2):163–188, 1991.
13. S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical Program Analysis Using General Purpose Logic Programming Systems — A Case Study. In *Programming Language Design and Implementation*, pages 117–126. ACM Press, 1996.
14. B. De Backer and H. Beringer. A CLP Language Handling Disjunctions of Linear Constraints. In *International Conference on Logic Programming*, pages 550–563. MIT Press, 1993.
15. C. Fecht. *Abstrakte Interpretation logischer Programme: Theorie, Implementierung, Generierung.* PhD thesis, Universität des Saarlandes, 1997.
16. M. García de la Banda, M. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens, and W. Simoens. Global Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 18(5):564–614, 1996.
17. M. Hermenegildo, R. Warren, and S. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*, 13(4):349–366, 1992.
18. A. King, J.-G. Smaus, and P. Hill. Quotienting Share for Dependency Analysis. In *European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 59–73. Springer, 1999.
19. B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems*, 16(1):35–101, 1994.
20. G. Plotkin. A Note on Inductive Generalisation. *Machine Intelligence*, 5:153–163, 1970.
21. P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Evaluation of the domain *Prop*. *Journal of Logic Programming*, 23(3):237–278, 1995.
22. J. Wunderwald. Memoing Evaluation by Source-to-Source Transformation. In *Logic Program Synthesis and Transformation*, Lecture Notes in Computer Science, pages 17–32. Springer, 1995. 1048.

The Correctness of Type Specialisation

John Hughes

Chalmers University of Technology, S-41296 Göteborg.
rjmh@cs.chalmers.se, www.cs.chalmers/~rjmh.

Abstract. Type specialisation, like partial evaluation, is an approach to specialising programs. But type specialisation works in a very different way, using a form of type inference. Previous articles have described the method and demonstrated its power as a program transformation, but its correctness has not previously been addressed. Indeed, it is not even clear what correctness should mean: type specialisation transforms programs to others with different types, so clearly cannot preserve semantics in the usual sense.
In this paper we explain why finding a correctness proof was difficult, we motivate a correctness condition, and we prove that type specialisation satisfies it. Perhaps unsurprisingly, type-theoretic methods turned out to crack the nut.

1 Introduction

Type specialisation, like partial evaluation, is an approach to specialising programs [13]. While partial evaluation focusses on specialising the control structures of a program, type specialisation focusses on transforming the datatypes. A type specialiser can produce programs operating on quite different types from the source program, and as a result achieve very strong specialisations. Earlier papers contain many illustrations of the power of the method [10,9,12,11,4].

However, these earlier papers do not address the *correctness* of the method: are the programs which type specialisation produces equivalent to those they are derived from? This question is harder to answer for type specialisation than for partial evaluation for two reasons. Firstly, since the type specialiser changes types, it is not even clear what 'equivalent' means. Secondly, for the most part, a partial evaluator applies a sequence of small semantics preserving transformations whose correctness is obvious, but the type specialiser is described by axiomatising the relation between source and residual programs in one go. Thus there is more scope for error. Indeed, it transpires that the type specialiser does *not* preserve semantics, but we are able to prove a weaker result which is 'good enough'.

In this paper, we present our proof of correctness. We shall begin by reviewing type specialisation, and explaining the problems which foiled our earlier attempts to find a proof. Then we explain what we actually prove, which is an analogue of subject reduction. Finally, we will present some of the cases of the proof in detail.

G. Smolka (Ed.): ESOP/ETAPS 2000, LNCS 1782, pp. 215–229, 2000.

2 What is Type Specialisation?

Type specialisation transforms a typed source program into a typed residual program, and in constrast to partial evaluation, types play a major rôle during the transformation itself. Both source and residual programs are simply typed, but they are expressed in different languages, and their types are used for different purposes. In Figure 1 we specify the syntax of terms and types for a small language we will study first.

$$
\begin{array}{rcl}
e & ::= & n \mid e + e \\
 & \mid & \underline{\textbf{lift}}\ e \\
 & \mid & x \mid \underline{\lambda} x.e \mid e \underline{@} e \\
 & \mid & \underline{\textbf{fix}}\ e \\
 & & \\
\tau & ::= & \textbf{int} \\
 & \mid & \underline{\textbf{int}} \\
 & \mid & \tau \underline{\rightarrow} \tau
\end{array}
\qquad
\begin{array}{rcl}
e' & ::= & \bullet \\
 & \mid & n \\
 & \mid & x \mid \lambda x.e' \mid e'\ e' \\
 & \mid & \textbf{fix}\ e' \\
 & & \\
\tau' & ::= & n \\
 & \mid & \textbf{int} \\
 & \mid & \tau' \rightarrow \tau'
\end{array}
$$

Fig. 1. Source and Residual Languages.

The source language is a form of two-level λ-calculus: constructions may come in two forms, static or dynamic, with the dynamic form indicated by underlining. Similarly, types may be either static or dynamic. In the figure, we consider only static integers (constants or additions), dynamic integers (formed by applying $\underline{\textbf{lift}}$ to static ones), and dynamic functions ($\underline{\lambda}$-expressions, dynamic application, and dynamic **fix**). The typing rules for this fragment are given in Figure 2.

$$
\Gamma \vdash n : \textbf{int}
\qquad
\frac{\Gamma \vdash e : \textbf{int}}{\Gamma \vdash \underline{\textbf{lift}}\ e : \underline{\textbf{int}}}
\qquad
\frac{\Gamma \vdash e_1 : \tau_1 \underline{\rightarrow} \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 \underline{@} e_2 : \tau_2}
$$

$$
\frac{\Gamma \vdash e_i : \textbf{int}}{\Gamma \vdash e_1 + e_2 : \textbf{int}}
\qquad
\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \underline{\lambda} x.e : \tau_1 \underline{\rightarrow} \tau_2}
\qquad
\frac{\Gamma \vdash e : \tau \underline{\rightarrow} \tau}{\Gamma \vdash \underline{\mathit{fix}}\ e : \tau}
$$

$$
\Gamma, x : \tau \vdash x : \tau
$$

Fig. 2. Source Typing Rules.

There are two subtleties here, however. Firstly, in contrast to other two-level λ-calculi, we do not restrict the formation of two level types in any way. For example, we allow dynamic functions to take static values as arguments, and return static results, which is forbidden in the context of partial evaluation.

The reason is simply that the type specialiser is able to specialise such programs, while partial evaluators are not. Intuitions from other specialisers lead one astray here therefore: the reason that no restrictions on type formation are stated is not that I have forgotten them, but that there are indeed no restrictions.

Secondly, we interpret the syntax of types co-inductively. That is, types may be *infinite* expressions conforming to this syntax. This is the way in which we handle recursive types: they are represented as their infinite unfolding, and no special construction for type recursion is required. This is particularly useful for residual types, since it allows the specialiser to synthesize recursive types freely. While recursive types correspond only to *regular* infinite types, we need no assumption of regularity in the proofs which follow. Recursive types are of little use in the fragment in the Figure, but when we later extend the language we consider they will of course play their usual useful rôle.

The residual language is also a form of simply typed λ-calculus, but with a rich type system in which types carry static information. Thus there is a residual type n for every integer n; a static integer expression in the source language specialises to a residual expression with such a type. All static information is expressed via residual types, and as a result need not be present in residual terms. This explains the residual term $\bullet$, which stands for 'no value': we can specialise $2+2$ for example to $\bullet$, since the residual type (4) already tells us all we need to know about the result. Type specialisation produces many residual expressions of this sort, but they are easy to remove in a post-processor we call the 'void eraser'. The typing rules for residual terms are given in Figure 3.

$$\Gamma \vdash \bullet : n \qquad\qquad \Gamma \vdash n : \mathbf{int} \qquad\qquad \Gamma, x : \tau' \vdash x : \tau'$$

$$\frac{\Gamma, x : \tau'_1 \vdash e : \tau'_2}{\Gamma \vdash \lambda x.e : \tau'_1 \to \tau'_2} \qquad \frac{\Gamma \vdash e'_1 : \tau'_1 \to \tau'_2 \quad \Gamma \vdash e'_2 : \tau'_1}{\Gamma \vdash e'_1\ e'_2 : \tau'_2} \qquad \frac{\Gamma \vdash e' : \tau' \to \tau'}{\Gamma \vdash \mathbf{fix}\ e' : \tau'}$$

Fig. 3. Residual Typing Rules.

Type specialisation is specified via a set of specialisation rules, analogous to typing rules. Specialisation rules let us infer specialisation judgements, of the form

$$\Gamma \vdash e : \tau \hookrightarrow e' : \tau'$$

meaning that source expression e of type τ specialises to residual expression e' of type τ'. The context Γ contains assumptions about source variables, of the form

$$x : \tau \hookrightarrow e' : \tau'$$

Notice that variables may specialise to any residual expression; they do not have to specialise to variables.

$$\Gamma \vdash n : \mathbf{int} \hookrightarrow \bullet : n$$

$$\frac{\Gamma \vdash e_i : \mathbf{int} \hookrightarrow e'_i : n_i \qquad n = n_1 + n_2}{\Gamma \vdash e_1 + e_2 : \mathbf{int} \hookrightarrow \bullet : n}$$

$$\frac{\Gamma \vdash e : \mathbf{int} \hookrightarrow e' : n}{\Gamma \vdash \underline{\mathbf{lift}}\ e : \underline{\mathbf{int}} \hookrightarrow n : \mathbf{int}}$$

$$\Gamma, x : \tau \hookrightarrow e' : \tau' \vdash x : \tau \hookrightarrow e' : \tau'$$

$$\frac{\Gamma, x : \tau_1 \hookrightarrow x' : \tau'_1 \vdash e : \tau_2 \hookrightarrow e' : \tau'_2}{\Gamma \vdash \underline{\lambda} x.e : \tau_1 \underline{\to} \tau_2 \hookrightarrow \lambda x'.e' : \tau'_1 \to \tau'_2} \quad x' \notin FV(\Gamma)$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \underline{\to} \tau_2 \hookrightarrow e'_1 : \tau'_1 \to \tau'_2 \qquad \Gamma \vdash e_2 : \tau_1 \hookrightarrow e'_2 : \tau'_1}{\Gamma \vdash e_1 \underline{@} e_2 : \tau_2 \hookrightarrow e'_1\ e'_2 : \tau'_2}$$

$$\frac{\Gamma \vdash e : \tau \underline{\to} \tau \hookrightarrow e' : \tau' \to \tau'}{\Gamma \vdash \underline{\mathbf{fix}}\ e : \tau \hookrightarrow \mathbf{fix}\ e' : \tau'}$$

Fig. 4. Specialisation Rules.

The specialisation rules for the fragment we are considering here are given in Figure 4. Using these rules we can conclude, for example, that

$$\vdash (\underline{\lambda} x.\underline{\mathbf{lift}}\ (x+1))\underline{@}2 : \underline{\mathbf{int}} \hookrightarrow (\lambda x'.3)\ \bullet : \mathbf{int}$$

The $2 : \mathbf{int}$ specialises to $\bullet : 2$, which forces the type of x' to be 2. Consequently $x + 1 : \mathbf{int}$ specialises to $\bullet : 3$, and the $\underline{\mathbf{lift}}$ moves this static information back into the term, specialising to $3 : \mathbf{int}$. Void erasure in this case elides both $\bullet$ and $\lambda x'$, resulting in just 3 as the final specialised program.

Note that the residual type system is more restrictive than the source one, so that well-typed source programs may fail to specialise. For example, the term

$$(\underline{\lambda} f.f\underline{@}2 + f\underline{@}3)\underline{@}(\underline{\lambda} x.x + 1)$$

cannot be specialised, because x would need to be assigned both residual types 2 and 3. This is perfectly natural: when we introduce the possibility to specialise types, we also introduce the possibility to do so inconsistently at different points in the program.

Using types to carry static information enables us to specialise more programs than a partial evaluator can. For example,

$$\vdash (\underline{\lambda} f.f\underline{@}2)\underline{@}(\underline{\lambda} x.\underline{\mathbf{lift}}\ (x+1)) : \underline{\mathbf{int}} \hookrightarrow (\lambda f'.f'\ \bullet)\ (\lambda x'.3) : \mathbf{int}$$

where x' must have type 2 to match the call of f', and so the body of f' specialises to 3. Here we can specialise the body of $\underline{\lambda} x.\underline{\mathbf{lift}}\ (x+1)$, even though it does not appear in an application. A partial evaluator, such as λ-MIX [6] or Similix [2], would need to contract at least the outer β-redex in order to propagate a static argument to x, but since this is a dynamic β-redex then this is forbidden; this program is not well annotated for partial evaluation, but causes the type specialiser no problems. In larger programs where it is important not to unfold certain function calls, then this capability gives the type specialiser substantially more power.

There is much more to the type specialiser than this, but we will introduce further features later, along with their proofs of correctness.

3 Why is Correctness Difficult?

Of course, we would like to know that specialisation does not change the semantics of programs; residual programs should be equivalent to the source programs they were derived from. Yet we cannot hope to prove this for the type specialiser. The very essence of the type specialiser is that it changes types. The source and residual programs in general have quite different types, and so they lie in different semantic domains: we certainly cannot expect them to be equal. For example, 42 specialises to $\bullet$, and of course these are different.

However, we note that dynamic type constructors always specialise to one-level versions of themselves — in our fragment this refers to $\underline{\textbf{int}}$ and $\underline{\to}$. Thus, if the type of an expression involves only these constructors, then it will specialise to a residual expression with an isomorphic type. Thus we might hope to prove equivalence in this case.

Unfortunately, it doesn't hold. Consider the source term $\underline{\textbf{lift}}\ (\underline{\textbf{fix}}\ (\underline{\lambda}x.x))$, which clearly denotes $\perp$. If we assume $x : \textbf{int} \hookrightarrow x' : 42$, then we can specialise $\underline{\lambda}x.x$ to $\lambda x'.x' : 42 \to 42$, and so specialise the fixpoint to a term with type 42. Now the rule for $\underline{\textbf{lift}}$ lets us specialise the entire term to $42 : \textbf{int}$, which is clearly not equivalent to the source expression. In this case the implemented specialiser would not actually choose this specialisation, but we can force it to exhibit similar behaviour by supplying slightly more complex terms. For example,

$$\underline{\textbf{lift}}\ (\underline{\textbf{fix}}\ (\underline{\lambda}x.\underline{\textbf{if}}\ \underline{\textbf{true}}\ \underline{\textbf{then}}\ x\ \underline{\textbf{else}}\ 42))$$

specialises to 42, but denotes $\perp$.

Instead of equivalence, therefore, we will aim to prove that the source term approximates the residual one. That is, the type specialiser may transform non-terminating programs into terminating ones, but it will never transform a terminating program into one which produces a different answer. Many program transformations behave similarly, so we will consider this weaker correctness property to be acceptable.

4 Outline of the Proof

Since type specialisation is modelled closely on type inference, it is perhaps not so surprising that type theoretic methods turn out to be useful. We will prove the correctness of the specialiser by showing a kind of subject reduction result. We will define source and residual reduction relations, both of which we write as $\rightarrow$, and then we will prove

> **Theorem** (Simulation). If $\Gamma \models e_1 : \tau \hookrightarrow e'_1 : \tau'$ and $e_1 \rightarrow e_2$, then there exists an e'_2 such that $\Gamma \models e_2 : \tau \hookrightarrow e'_2 : \tau'$ and $e'_1 \rightarrow^* e'_2$.

By this theorem we know that if e_1 eventually reduces to a value, then e'_1 reduces to the specialisation of a value. By

> **Lemma** (Value Specialisation). If $\Gamma \models v : \tau \hookrightarrow e' : \tau'$ (where v is a source value), then e' is a residual value.

then the following correctness theorem follows:

> **Theorem** (Correctness). If $\Gamma \models e : \tau \hookrightarrow e' : \tau'$ and e reduces to a value, then so does e'.

In order to prove the Simulation theorem, then we will need two lemmata about substitution — two, because we have two kinds of variables, and therefore two kinds of substitution. The lemma for source substitution is

> **Lemma** (Source Substitution). If $\Gamma \models e_1 : \tau_1 \hookrightarrow e'_1 : \tau'_1$ and $\Gamma, x : \tau_1 \hookrightarrow e'_1 : \tau'_1 \models e_2 : \tau_2 \hookrightarrow e'_2 : \tau'_2$, then $\Gamma \models e_2[e_1/x] : \tau_2 \hookrightarrow e'_2 : \tau'_2$.

No substitution is required into the residual term, because specialisation itself substitutes e'_1 for x.

The residual subsitution lemma is even simpler.

> **Lemma** (Residual Substitution). Let θ be a substition of residual terms for the residual variables occurring free in Γ. If $\Gamma \models e : \tau \hookrightarrow e' : \tau'$, then $\Gamma\theta \models e : \tau \hookrightarrow e'\theta : \tau'$.

We prove both these lemmata, and the Simulation theorem, by induction over the structure of source terms. In the next section we present the proofs for the fragment we are currently consideration, and then in later sections we show the cases for extensions to this fragment.

5 The Correctness of the Fragment

Before we go further we must define reduction relations for the source and target languages. We do so in Figure 5; the reduction relations are the smallest congruences satisfying the stated properties. By a *value* we mean a closed weak head normal form: the values in the source language take the form n, **lift** n, or $\underline{\lambda} x.e$, while the values in the residual language take the form $\bullet$, n or $\lambda \overline{x}.e'$. The Value Specialisation lemma now follows directly, by applying the appropriate specialisation rule to each form of source value. We now prove the substitution lemmata and the Simulation theorem in turn.

$$\begin{array}{lll} n_1 + n_2 & \to n & \text{if } n = n_1 + n_2 \\ (\underline{\lambda} x.e_1)\underline{@}e_2 & \to e_1[e_2/x] & \\ \underline{\mathbf{fix}}\ e & \to e\underline{@}(\underline{\mathbf{fix}}\ e) & \end{array} \qquad \begin{array}{ll} & \\ (\lambda x.e_1')\ e_2' & \to e_1'[e_2'/x] \\ \mathbf{fix}\ e' & \to e'\ (\mathbf{fix}\ e') \end{array}$$

Fig. 5. Source and Residual Reduction Rules.

Proof of the Source Substitution Lemma. We are to prove that if $\Gamma \vdash e_1 : \tau_1 \hookrightarrow e_1' : \tau_1'$ and $\Gamma, x : \tau_1 \hookrightarrow e_1' : \tau_1' \vdash e_2 : \tau_2 \hookrightarrow e_2' : \tau_2'$, then $\Gamma \vdash e_2[e_1/x] : \tau_2 \hookrightarrow e_2' : \tau_2'$. The proof is by induction over the syntax of e_2. The only interesting case is that for variables. For the variable x, we must show that

$$\Gamma \vdash x[e_1/x] : \tau_2 \hookrightarrow e_2' : \tau_2'$$

But from the second assumption, we know that

$$\Gamma, x : \tau_1 \hookrightarrow e_1' : \tau_1' \vdash x : \tau_2 \hookrightarrow e_2' : \tau_2'$$

Consulting the specialisation rule for variables, it follows that e_1' and e_2' are the same, as are τ_1 and τ_2, and τ_1' and τ_2'. Since by the first assumption,

$$\Gamma \vdash e_1 : \tau_1 \hookrightarrow e_1' : \tau_1'$$

then the result follows. For other variables, the proof is trivial.

Proof of the Residual Substitution Lemma. We are to prove that if θ is a substition of residual terms for residual variables, and $\Gamma \vdash e : \tau \hookrightarrow e' : \tau'$, then $\Gamma\theta \vdash e : \tau \hookrightarrow e'\theta : \tau'$. Once again the proof is by induction on the syntax of e. We will prove the cases for variables and $\underline{\lambda}$-expressions, since these are the only rules that can introduce residual variables into the residual term.

For a variable x, we assume that $\Gamma \vdash x : \tau \hookrightarrow e' : \tau'$, which by the specialisation rule for variables means that Γ must contain an assumption of the form $x : \tau \hookrightarrow e' : \tau'$. $\Gamma\theta$ therefore contains the assumption $x : \tau \hookrightarrow e'\theta : \tau'$, and it follows that $\Gamma\theta \vdash x : \tau \hookrightarrow e'\theta : \tau'$ as required.

For a $\underline{\lambda}$-expression $\underline{\lambda} x.e$, we know that its specialisation uses the rule

$$\frac{\Gamma, x : \tau_1 \hookrightarrow x' : \tau_1' \vdash e : \tau_2 \hookrightarrow e' : \tau_2'}{\Gamma \vdash \underline{\lambda} x.e : \tau_1 \underline{\to} \tau_2 \hookrightarrow \lambda x'.e' : \tau_1' \to \tau_2'} \quad x' \notin FV(\Gamma)$$

Since x' is not free in Γ, it cannot be renamed by θ, so we may conclude by the induction hypothesis that

$$\Gamma\theta, x : \tau_1 \hookrightarrow x' : \tau_1' \vdash e : \tau_2 \hookrightarrow e'\theta : \tau_2'$$

Applying the specialisation rule for $\underline{\lambda}$ again, we derive

$$\Gamma\theta \vdash \underline{\lambda} x.e : \tau_1 \underline{\to} \tau_2 \hookrightarrow (\lambda x'.e')\theta : \tau_1' \to \tau_2'$$

as required.

Proof of the Simulation Theorem . We are to prove that if $\Gamma \vdash e_1 : \tau \hookrightarrow e_1' : \tau'$ and $e_1 \to e_2$, then there exists an e_2' such that $\Gamma \vdash e_2 : \tau \hookrightarrow e_2' : \tau'$ and $e_2 \to^* e_2'$. This proof is also by induction on the syntax of e_1, and we will present it in some detail.

- Case n. Trivial, since n does not reduce to anything.
- Case $e_1 + e_2$. According to the specialisation rule for +, we have

  $$\frac{\Gamma \vdash e_i : \mathbf{int} \hookrightarrow e_i' : n_i \qquad n = n_1 + n_2}{\Gamma \vdash e_1 + e_2 : \mathbf{int} \hookrightarrow \bullet : n}$$

  Suppose first that e_1 and e_2 are both values. Since

  $$\Gamma \vdash e_1 : \mathbf{int} \hookrightarrow e_1' : n_1$$

  then e_1 must be n_1, and similarly for e_2. It follows that $e_1 + e_2 \to n$, which specialises to $\bullet : n$. It remains to show that $\bullet \to^* \bullet$, which it does in zero steps.
  Alternatively, suppose without loss of generality that $e_1 + e_2 \to e_3 + e_2$ by reducing $e_1 \to e_3$. Then by the induction hypothesis, there is an e_3' such that $e_1' \to^* e_3'$ and

  $$\Gamma \vdash e_3 : \mathbf{int} \hookrightarrow e_3' : n_1$$

  Applying the specialisation rule for +, we derive

  $$\Gamma \vdash e_3 + e_2 : \mathbf{int} \hookrightarrow \bullet : n$$

  and it remains only to show $\bullet \to^* \bullet$ as before.
- Case $\underline{\mathbf{lift}}\ e$. We have $\underline{\mathbf{lift}}\ e \to \underline{\mathbf{lift}}\ e_0$, and

  $$\frac{\Gamma \vdash e : \mathbf{int} \hookrightarrow e' : n}{\Gamma \vdash \underline{\mathbf{lift}}\ e : \underline{\mathbf{int}} \hookrightarrow n : \mathbf{int}}$$

  We have $e \to e_0$, and so by the induction hypothesis there is an e_0' such that $e' \to^* e_0'$ and $\Gamma \vdash e_0 : \mathbf{int} \hookrightarrow e_0' : n$. It follows that

  $$\Gamma \vdash \underline{\mathbf{lift}}\ e_0 : \underline{\mathbf{int}} \hookrightarrow n : \mathbf{int}$$

  and since $n \to^* n$ then the proof is complete.
- Case x. Trivial since there is no reduction rule for variables.
- Case $\underline{\lambda} x.e$. We have $\underline{\lambda} x.e \to \underline{\lambda} x.e_0$, and

  $$\frac{\Gamma, x : \tau_1 \hookrightarrow x' : \tau_1' \vdash e : \tau_2 \hookrightarrow e' : \tau_2'}{\Gamma \vdash \underline{\lambda} x.e : \tau_1 \underline{\to} \tau_2 \hookrightarrow \lambda x'.e' : \tau_1' \to \tau_2'} \quad x' \notin FV(\Gamma)$$

  So $e \to e_0$, and by the induction hypothesis there is an e_0' such that $e' \to^* e_0'$ and

  $$\Gamma, x : \tau_1 \hookrightarrow x' : \tau_1' \vdash e_0 : \tau_2 \hookrightarrow e_0' : \tau_2'$$

  It follows that

  $$\Gamma \vdash \underline{\lambda} x.e_0 : \tau_1 \underline{\to} \tau_2 \hookrightarrow \lambda x'.e_0' : \tau_1' \to \tau_2'$$

  and $\lambda x'.e' \to^* \lambda x'.e_0'$ as required.

- Case $e_1 \underline{@} e_2$. An application can be reduced in three different ways: a reduction may be made inside e_1, or inside e_2, or the application itself may be a β-redex which is reduced. The first two cases are proved in the same way as the $\underline{\lambda}$ case above, so we consider only the third. Suppose therefore that e_1 is $\underline{\lambda} x.e$. Combining the specialisation rules for $\underline{\lambda}$ and $\underline{@}$, we obtain

$$\dfrac{\dfrac{\Gamma, x : \tau_1 \hookrightarrow x' : \tau_1' \vdash e : \tau_2 \hookrightarrow e' : \tau_2'}{\Gamma \vdash \underline{\lambda} x.e : \tau_1 \underline{\rightarrow} \tau_2 \hookrightarrow \lambda x'.e' : \tau_1' \rightarrow \tau_2'} \quad \Gamma \vdash e_2 : \tau_1 \hookrightarrow e_2' : \tau_1'}{\Gamma \vdash (\underline{\lambda} x.e) \underline{@} e_2 : \tau_2 \hookrightarrow (\lambda x'.e')\ e_2' : \tau_2'}$$

Substituting e_2' for x' using the Residual Substitution lemma, we know that

$$\Gamma, x : \tau_1 \hookrightarrow e_2' : \tau_1' \vdash e : \tau_2 \hookrightarrow e'[e_2'/x'] : \tau_2'$$

Now by the Source Substitution lemma, we have

$$\Gamma \vdash e[e_2/x] : \tau_2 \hookrightarrow e'[e_2'/x'] : \tau_2'$$

Since $(\underline{\lambda} x.e) \underline{@} e_2 \rightarrow e[e_2/x]$ and $(\lambda x'.e')\ e_2' \rightarrow e'[e_2'/x']$, then the proof of this case is complete.
- Case $\underline{\textbf{fix}}\ e$. This case is similar to application, and is omitted.

This completes the proof of the Simulation theorem for the fragment.

6 Extensions

The tiny language we have considered so far illustrates only the basics of type specialisation: it consists only of dynamic λ-calculus plus one kind of static information. In reality the type specialiser accepts a much richer language. In this section we discuss some of the extensions, and their proofs of correctness.

6.1 Enriching the Dynamic Language

In addition to dynamic function types with dynamic $\underline{\lambda}$-expressions and applications, the type specialiser supports dynamic product types with tuples and selectors, dynamic tagged sum types with constructor application and a $\underline{\textbf{case}}$ expression, dynamic $\underline{\textbf{let}}$ expressions and conditionals. In each case we add a dynamic version of each construct to the source language, and a residual version to the residual language. The new reduction rules in the source and residual language correspond. Each dynamic construct specialises to its corresponding residual construct, with specialised sub-expressions. The substitution lemmata extend easily, and the proofs of the Simulation theorem all take the same form: a reduction in a sub-expression is simulated by reductions in the corresponding residual sub-expression, while a reduction using a new source reduction rule is simulated using the corresponding new residual reduction rule. The proofs are modelled on those for $\underline{\lambda} x.e$ and $e_1 \underline{@} e_2$.

6.2 Static Tagged Sums

One of the most interesting applications of the type specialiser is to remove type tags when specialising an interpreter for a typed language. If such an interpreter represents values using a universal type which is a tagged sum of the differently typed alternatives, then the type specialiser can remove the tags, specialising the universal type to an appropriate representation type at each use. To express this, we must add static tagged sum types to our source language. We extend the syntax of types and expressions as follows, where C is a tag, or 'constructor':

$$\begin{array}{rl} \tau ::= & \Sigma_{i=1}^{n}\ C\ \tau \\ e ::= & C\ e \\ \mid & \textbf{case}\ e\ \textbf{of}\,\{C\ x \rightarrow e\}_{i=1}^{n}\,\textbf{end} \end{array}$$

Since the tags are static, the corresponding residual types must record which constructor was actually applied. Thus we extend residual types as follows:

$$\tau' ::= C\ \tau'$$

There is no need to extend the language of residual terms, since application and inspection of static constructors will be specialised away.

The specialisation rule for a constructor application just records the constructor in the residual type,

$$\frac{\Gamma \vdash e : \tau_k \hookrightarrow e' : \tau'_k}{\Gamma \vdash C_k\ e : \Sigma_{i=1}^{n}\ C_i\ \tau_i \hookrightarrow e' : C_k\ \tau'_k}$$

while the rule for a **case** expression uses the statically-known constructor to choose the corresponding branch:

$$\frac{\begin{array}{c}\Gamma \vdash e : \Sigma_{i=1}^{n}\ C_i\ \tau_i \hookrightarrow e' : C_k\ \tau'_k \\ \Gamma, x_k : \tau_k \hookrightarrow e' : \tau'_k \vdash e_k : \tau_0 \hookrightarrow e'_k : \tau'_0\end{array}}{\Gamma \vdash \textbf{case}\ e\ \textbf{of}\,\{C_i\ x_i \rightarrow e_i\}_{i=1}^{n}\,\textbf{end} : \tau_0 \hookrightarrow e'_k : \tau'_0}$$

The Source and Residual substitution lemmata extend easily to these cases.

There is one new source reduction rule, namely

$$\textbf{case}\ C_k\ e\ \textbf{of}\,\{C_i\ x_i \rightarrow e_i\}_{i=1}^{n}\,\textbf{end} \rightarrow e_k[e/x_k]$$

and one new form of source value: $C\ v$. Notice that in order to prove the Value Specialisation lemma, we must require the argument of the constructor to be evaluated.

We will prove just the case in the Simulation theorem when the new reduction rule is applied. Thus we must prove that if

$$\Gamma \vdash \textbf{case}\ C_k\ e\ \textbf{of}\,\{C_i\ x_i \rightarrow e_i\}_{i=1}^{n}\,\textbf{end} : \tau_0 \hookrightarrow e'_k : \tau'_0$$

then there is an e'' such that $e'_k \rightarrow^* e''$ and

$$\Gamma \vdash e_k[e/x_k] : \tau_0 \hookrightarrow e'' : \tau'_0$$

We shall take e'' to be just e'_k, and argue that from the assumption we know that

$$\Gamma, x_k : \tau_k \hookrightarrow e' : \tau'_k \vdash e_k : \tau_0 \hookrightarrow e'_k : \tau'_0$$

where

$$\Gamma \vdash e : \tau_k \hookrightarrow e' : \tau'_k$$

By the Source Substitution lemma, it follows that $\Gamma \vdash e_k[e/x_k] : \tau_0 \hookrightarrow e'_k : \tau'_0$ as required.

6.3 Polyvariance

All interesting program specialisers are *polyvariant*, that is, they can specialise one expression in the source code multiple times. Polyvariance is provided in the type specialiser by extending the source and residual languages as follows:

$$\begin{array}{ll} e ::= \mathbf{poly}\ e \mid \mathbf{spec}\ e & \quad e' ::= (e', \ldots, e') \mid \pi_k\ e' \\ \tau ::= \mathbf{poly}\ \tau & \quad \tau' ::= (\tau', \ldots, \tau') \end{array}$$

The idea is that **poly** e can be specialised to a tuple of specialisations of e, from which **spec** e chooses an element. The residual type of such a tuple records which specialisations it contains. We add reduction rules

$$\mathbf{spec}\ (\mathbf{poly}\ e) \to e \qquad \pi_k\ (e'_1, \ldots, e'_n) \to e'_k$$

and new source values **poly** e, and residual values $(e'_1, \ldots, e'_n)$.

The specialisation rules for these constructions are:

$$\frac{\Gamma \vdash e : \tau \hookrightarrow e'_i : \tau'_i, i = 1 \ldots n}{\Gamma \vdash \mathbf{poly}\ e : \mathbf{poly}\ \tau \hookrightarrow (e'_1, \ldots, e'_n) : (\tau'_1, \ldots, \tau'_n)}$$

$$\frac{\Gamma \vdash e : \mathbf{poly}\ \tau \hookrightarrow e' : (\tau'_1, \ldots, \tau'_n)}{\Gamma \vdash \mathbf{spec}\ e : \tau \hookrightarrow \pi_k\ e' : \tau'_k}$$

The proofs of the substitution lemmata and the Simulation theorem go through easily for this extension. For the Simulation theorem, a reduction **poly** $e_1 \to$ **poly** e_2 by $e_1 \to e_2$ can be simulated by reductions in *each* specialisation, while the reduction **spec** (**poly** e) $\to e$ is simulated by $\pi_k\ (e'_1, \ldots, e'_n) \to e'_k$.

6.4 Static Functions

All interesting specialisers provide *static functions*, that is, functions which are unfolded at specialisation time. So, too, does the type specialiser, via static λ-expressions and static applications. The specialisation rule for static λ given in [10] is

$$\begin{array}{l} \{x_i : \tau_i \hookrightarrow e'_i : \tau'_i\} \vdash \\ \qquad \lambda x.e : \tau_a \to \tau_b \hookrightarrow (e'_1, \ldots, e'_n) : \mathbf{close}\ \{x_i : \tau_i \hookrightarrow \tau' : i\}\ \mathbf{in}\ \lambda x.e \end{array}$$

That is, a static function is represented in the residual program by a tuple of its free variables, and the residual type carries the information needed to unfold β-redexes.

Unfortunately, this specialisation rule violates the theorems we are trying to prove. The Simulation Theorem fails because a reduction under a static λ changes the residual *type* of the specialisation; since residual reduction does not change types, it is impossible for the result of specialisation before the reduction to reduce to the result of specialisation afterwards. The Source Substitution lemma also fails, because a static λ specialises to a tuple with one element per free variable; substituting for one of these variables changes the size of the tuple. The problem here is that type specialisation essentially performs a closure conversion, and closure conversion is not preserved by substitution.

Our solution is to instead consider specialisation of closure-converted programs. Thus we prove the correctness of a variant of the type specialisation previously described. We extend the source and target languages as follows,

$$\begin{array}{ll} e ::= \mathbf{close}\ \{x = e\}^*\ \mathbf{in}\ \lambda x.e \mid e@e & \qquad e' ::= (e', \ldots, e') \mid \pi_k\ e' \\ \tau ::= \tau \to \tau & \qquad \tau' ::= \mathbf{close}\ \{x : \tau \hookrightarrow \tau'\}^*\ \mathbf{in}\ \lambda x.e \end{array}$$

with the restriction that all the free variables of $\lambda x.e$ must be bound in the associated definitions. Closures and residual tuples are both values.

We add a β reduction rule to the source language,

$$(\mathbf{close}\ \{x_i = e_i\}\ \mathbf{in}\ \lambda x.e)@e_x \to e[e_i/x_i, e_x/x]$$

and a rule for reducing projections to the residual language. Moreover, we forbid reduction of the body of a static λ – the only reductions of closures take place in the bindings $\{x_i = e_i\}$.

The specialisation rules for static closures and applications are

$$\frac{\Gamma \vdash e_i : \tau_i \hookrightarrow e'_i : \tau'_i, i \in 1 \ldots n}{\begin{array}{l}\Gamma \vdash \mathbf{close}\ \{x_i = e_i\}\ \mathbf{in}\ \lambda x.e : \tau_1 \to \tau_2 \hookrightarrow \\ \qquad (e'_1, \ldots, e'_n) : \mathbf{close}\ \{x_i : \tau_i \hookrightarrow \tau'_i\}\ \mathbf{in}\ \lambda x.e\end{array}}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_f : \tau_x \to \tau_y \hookrightarrow e'_f : \mathbf{close}\ \{x_i : \tau_i \hookrightarrow \tau'_i\}\ \mathbf{in}\ \lambda x.e \\ \Gamma \vdash e_x : \tau_x \hookrightarrow e'_x : \tau'_x \\ \{x_i : \tau_i \hookrightarrow \pi_i\ e'_f : \tau'_i\}, x : \tau_x \hookrightarrow e'_x : \tau'_x \vdash e : \tau_y \hookrightarrow e' : \tau'_y\end{array}}{\Gamma \vdash e_f@e_x : \tau_y \hookrightarrow e' : \tau'_y}$$

With these definitions, the substitution and value specialisation lemmata are easily proved. To prove the Simulation Theorem we must introduce another (easily proved) lemma:

> **Lemma** (Reduction in Context Lemma) Let Γ_2 be obtained from Γ_1 by making a reduction in one of the residual expressions. If $\Gamma_1 \vdash e : \tau \hookrightarrow e'_1 : \tau'$, then there exists e'_2 such that $e'_1 \to^* e'_2$ and $\Gamma_2 \vdash e : \tau \hookrightarrow e'_2 : \tau'$.

The proof of the Simulation Theorem now goes through.

7 Related Work

In 1991, Gomard and Jones described λ-MIX, the first self-applicable partial evaluator for the λ-calculus, which was so simple that much later work was based on it. Gomard proved the correctness of the partial evaluator, that is, that source and residual programs denote the same values [6]. The proof is based on establishing a logical relation between the denotation of a two-level source term, and the denotation of its one-level erasure. λ-MIX was the first partial evaluator whose binding-time analysis was expressed as a type system, and the logical relation is indexed by binding-time types.

Gomard's original proof was somewhat flawed by an informal treatment of fresh name generation. Moggi pointed this out, and gave a rigorous proof based on an alternative semantics for λ-MIX using functor categories [15]. Using related techniques, Filinksi has proved the soundness and completeness [5] of Danvy's type-directed partial evaluation (TDPE) [3].

These proofs have in common that they are based on establishing logical relations between denotational semantics of source and residual terms. This is essentially the approach we first tried to follow to show the correctness of the type specialiser. But since λ-MIX and TDPE do not transform types, the logical relations are simpler to define, and since neither allows recursive binding-time types, the problems they cause with well-definedness of logical relations do not arise. (Recursive types are not really needed in λ-MIX, since dynamic computations are essentially untyped).

Other recent work on the correctness of partial evaluators has focussed on the correctness of binding-time analysis, rather than on specialisation proper.

A closer analogy can be found with other recent work on type-directed transformations. John Hannan and Patrick Hicks have published a series of papers in which they present such transformations of higher order languages, for example [7,8]. Just like type specialisation, these transformations are specified by inference rules, whose judgements relate a source term, a transformed term, and a type in an extended type language specifying how the former should be transformed into the latter. Proofs of correctness are outlined, and are quite similar to our own: source and target languages are given an operational semantics, and there is an analogue of our Simulation Theorem relating the two. Hannan and Hicks also prove that every well-typed source term can be transformed to a target term, which is of course untrue for type specialisation, and that reductions of target terms can be simulated by the corresponding source terms.

8 Discussion and Conclusions

A first attempt to find a proof was based on giving a denotational semantics to source and target languages, and establishing a logical relation indexed by residual types between them. But this foundered when the relation proved to be ill-defined. The problem is that residual types may involve arbitrary type recursion under function arrows. A recursive type leads to a recursively defined logical relation, which only makes sense if the recursive definition has a least fixed point. But since the formation of logical relations on function types is antimonotonic in the left argument, then the usual monotonicity argument that a least fixed point exists does not apply.

It is possible that this approach might succeed even so. We could try to define a metric on relations, and show that the recursive definitions we are interested in are contractive, just as MacQueen, Plotkin and Sethi did to show that recursive types could be modelled by ideals [14]. But this would at best lead to a very technical proof, dependent on the detailed structure of the underlying semantic domains. Instead, we chose to pursue the more operational approach described in this paper.

The proof we have presented is pleasingly simple, and we have some hope that the proof method will be robust to extensions of the type specialiser, not least since similar methods have been used successfully to prove the correctness of other type-directed transformations. The operational approach, inspired by subject reduction, proved to be much easier to carry through the denotationally-based attempt. And of course, it is pleasing to know that type specialisation actually is correct.

The proof does raise other questions, though. For example, earlier papers were vague on whether the intended semantics of the object language was call-by-value or call-by-name. In this paper we explicitly give it a call-by-name semantics. Is type specialisation correct for a call-by-value language? One would hope that a similar proof would go through, but the most obvious idea of restricting β-reduction to β_v redexes does not seem to work easily. Another interesting idea would be to consider call-by-need reduction rules [1]: perhaps one could show thereby that specialisation (of a suitably restricted language) does not duplicate computations.

We have also focussed here on the relationship between source terms and residual *terms* – the dynamic part of the specialisation. Residual *types* in contrast play only a small rôle here. Yet we might also hope to be able to relate them to the source program. Residual types purport to carry static information about the source term they are derived from: in a sense they can be regarded as properties of source terms. For example, if $\vdash f : \mathbf{int} \underline{\rightarrow} \mathbf{int} \hookrightarrow f' : 42 \rightarrow 44$, then we would expect that f maps 42 to 44. Another interesting avenue would be to assign a semantics to residual types as properties, and prove that specialisation produces properties that really hold.

References

1. Zena Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. A call-by-need lambda calculus. In *22'nd Symposium on Principles of Programming Languages*, San Francisco, California, January 1995. ACM Press.
2. A. Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17(1-3):3–34, December 1991. Selected papers of ESOP '90, the 3rd European Symposium on Programming.

3. O. Danvy. Type-directed partial evaluation. In *Symposium on Principles of Programming Languages*. ACM, jan 1996.
4. D. Dussart, J. Hughes, and P. Thiemann. Type Specialisation for Imperative Languages. In *International Conference on Functional Programming*, pages 204–216, Amsterdam, June 1997. ACM.
5. Andrzej Filinski. A Semantic Account of Type-Directed Partial Evaluation. In *Principles and Practice of Declarative Programming: International Conference PPDP'99*, volume 1702 of *Lecture Notes in Computer Science*, pages 378–395, Paris, France, September 1999. Springer-Verlag.
6. C.K. Gomard. A self-applicable partial evaluator for the lambda calculus: Correctness and pragmatics. *ACM Transactions on Programming Languages and Systems*, 14(2):147–172, April 1992.
7. John Hannan and Patrick Hicks. Higher-Order Arity Raising. In *Proceedings of 3rd ACM SIGPLAN International Conference on Functional Programming*, pages 27–38, Baltimore, Maryland, September 1998.
8. John Hannan and Patrick Hicks. Higher-Order UnCurrying. In *Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–10, San Diego, January 1998.
9. J. Hughes. An Introduction to Program Specialisation by Type Inference. In *Functional Programming*. Glasgow University, July 1996. published electronically.
10. J. Hughes. Type Specialisation for the Lambda-calculus; or, A New Paradigm for Partial Evaluation based on Type Inference. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation*, volume 1110 of *LNCS*, pages 183–215. Springer-Verlag, February 1996.
11. J. Hughes. A Type Specialisation Tutorial. In *DIKU Summer School on Partial Evaluation*, 1998.
12. J. Hughes. Type Specialisation. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *1998 Symposium on Partial Evaluation*, volume 30 of *Computing Surveys*, September 1998.
13. N. D. Jones, , C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
14. David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71(1/2):95–130, October/November 1986.
15. Eugenio Moggi. Functor categories and two-level languages. In *FOSSACS*, volume 1378 of *Lecture Notes in Computer Science*, pages 211–225. Springer-Verlag, 1998.

Type Classes with Functional Dependencies[⋆]

Mark P. Jones

Department of Computer Science and Engineering
Oregon Graduate Institute of Science and Technology
Beaverton, Oregon, USA
mpj@cse.ogi.edu

Abstract. Type classes in Haskell allow programmers to define functions that can be used on a set of different types, with a potentially different implementation in each case. For example, type classes are used to support equality and numeric types, and for monadic programming. A commonly requested extension to support 'multiple parameters' allows a more general interpretation of classes as relations on types, and has many potentially useful applications. Unfortunately, many of these examples do not work well in practice, leading to ambiguities and inaccuracies in inferred types and delaying the detection of type errors.

This paper illustrates the kind of problems that can occur with multiple parameter type classes, and explains how they can be resolved by allowing programmers to specify explicit dependencies between the parameters. A particular novelty of this paper is the application of ideas from the theory of relational databases to the design of type systems.

1 Introduction

Type classes in Haskell [11] allow programmers to define functions that can be used on a set of different types, with a potentially different implementation in each case. Each class represents a set of types, and is associated with a particular set of *member functions*. For example, the type class *Eq* represents the set of all equality types, which is precisely the set of types on which the (==) operator can be used. Similarly, the type class *Num* represents the set of all numeric types—including *Int*, *Float*, complex and rational numbers—on which standard arithmetic operations like (+) and (−) can be used. These and several other classes are defined in the standard Haskell prelude and libraries [11, 12]. The language also allows programmers to define new classes or to extend existing classes to include new, user-defined datatypes. As such, type classes play an important role in many Haskell programs, both directly through uses of the member functions associated with a particular class, and indirectly in the use of various language constructs including a special syntax for monadic programming (the do-notation).

[⋆] The research reported in this paper was supported by the USAF Air Materiel Command, contract # F19628-96-C-0161.

G. Smolka (Ed.): ESOP/ETAPS 2000, LNCS 1782, pp. 230–244, 2000.

The use of type classes is reflected by allowing types to include *predicates*. For example, the type of the equality operator is written:

$$(==) :: Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool$$

The type variable a used here represents an arbitrary type (bound by an implicit universal quantifier), but the predicate $Eq\ a$ then restricts the possible choices for a to types that are in Eq. More generally, functions in Haskell have types of the form $P \Rightarrow \tau$, where P is some list of predicates and τ is a monotype. If P is empty, then we usually abbreviate $P \Rightarrow \tau$ as τ. In most implementations, the presence of a predicate in a function's type indicates that an implicit parameter will be added to pass some appropriate *evidence* for that predicate at run-time. For example, we might use an implementation of equality on values of type a as evidence for a predicate of the form $Eq\ a$. Details of this implementation scheme may be found elsewhere [14].

In a predicate such as $Eq\ a$, we refer to Eq as the *class name*, and to a as the *class parameter*. Were it not for the use of a restricted character set, constraints like this might instead have been written in the form $a \in Eq$, reflecting an intuition that Eq represents a set of types of which a is expected to be a member. The Haskell syntax, however, which looks more like a curried function application, suggests that it might be possible to allow classes to have more than one parameter. For example, what might a predicate of the form $R\ a\ b$ mean, where two parameters a and b have been provided? The obvious answer is to interpret R as a two-place relation between types, and to read $R\ a\ b$ as the assertion that a and b are related by R. This is a natural generalization of the one parameter case because sets are just one-place relations. More generally, we can interpret an n parameter class by an n-place relation on types.

One potential application for multiple parameter type classes was suggested (but not pursued) by Wadler and Blott in the paper where type classes were first described [14]. The essence of their example was to use a two parameter class *Coerce* to describe a subtyping relation, with an associated coercion operator:

$$coerce :: Coerce\ a\ b \Rightarrow a \rightarrow b.$$

In the decade since that paper was published, many other applications for multiple parameter type classes have been discovered [13]; we will see some of these in later sections of the current paper. The technical foundations for multiple parameter classes have also been worked out during that time, and support for multiple parameter type classes is now included in some of the currently available Haskell implementations. So it is perhaps surprising that support for multiple parameter type classes is still not included in the Haskell standard, even in the most recent revision [11]. One explanation for this reticence is that some of the proposed applications have not worked particularly well in practice. These problems often occur because the relations on types that we can specify using simple extensions of Haskell are too general for practical applications. In particular, they fail to capture important dependencies between parameters. More concretely, the use of multiple parameter classes can often result in ambiguities and inaccuracies in inferred types, and in delayed detection of type errors.

In this paper, we show that many of these problems can be avoided by giving programmers an opportunity to specify the desired relations on types more precisely. The key idea is to allow the definitions of type classes to be annotated with functional dependencies—an idea that originates in the theory of relational databases. In Section 2, we describe the key features of Haskell type classes that will be needed to understand the contributions of this paper. In Section 3, we use the design of a simple library of collection types to illustrate the problems that can occur with multiple parameter classes, and to motivate the introduction of functional dependencies. Further examples are provided in Section 4. Basic elements of the theory of functional dependencies are presented in Section 5, and are used to explain their role during type inference in Section 6. In Section 7, we describe some further opportunities for using dependency information, and then we conclude with some pointers to future work in Section 8.

2 Preliminaries: Type Classes in Haskell

This section describes the *class declarations* that are used to introduce new (single parameter) type classes in Haskell, and the *instance declarations* that are used to populate them. Readers who are already familiar with these aspects of Haskell should probably skip ahead to the next section. Those requiring more than the brief overview given here should refer to the Haskell report [11] or to the various tutorials and references listed on the Haskell website at `http://haskell.org`.

Class Declarations: A class declaration specifies the name for a class and lists the member functions that each type in the class is expected to support. The actual types in each class—which are normally referred to as the *instances* of the class—are described using separate declarations, as will be described below. For example, an *Eq* class, representing the set of equality types, might be introduced by the following declaration:

$$\begin{array}{l} \textbf{class}\ Eq\ a\ \textbf{where} \\ \quad (==) :: a \to a \to Bool \end{array}$$

The type variable a that appears in both lines here represents an arbitrary instance of the class. The intended reading of the declaration is that, if a is a particular instance of Eq, then we can use the $(==)$ operator at type $a \to a \to Bool$ to compare values of type a.

Qualified Types: As we have already indicated, the restriction on the use of the equality operator is reflected in the type that is assigned to it:

$$(==) :: Eq\ a \Rightarrow a \to a \to Bool$$

Types that are restricted by a predicate like this are referred to as *qualified types* [4]. Such types will be assigned to any function that makes either direct or indirect use of the member functions of a class at some unspecified type. For example, the functions:

```
member x xs = any (x ==) xs
subset xs ys  = all (\x → member x ys) xs
```

will be assigned types:

```
member :: Eq a ⇒ a → [a] → Bool
subset   :: Eq a ⇒ [a] → [a] → Bool.
```

Superclasses: Classes may be arranged in a hierarchy, and may have multiple member functions. The following example illustrates both with a declaration of the *Ord* class, which contains the types whose elements can be ordered using strict (<) and non-strict (<=) comparison operators:

```
class Eq a ⇒ Ord a where
  (<), (<=) :: a → a → Bool
```

In this particular context, the ⇒ symbol should not be read as implication; in fact reverse implication would be a more accurate reading, the intention being that every instance of *Ord* is also an instance of *Eq*. Thus *Eq* plays the role of a *superclass* of *Ord*. This mechanism allows the programmer to specify an expected relationship between classes: it is the compiler's responsibility to ensure that this property is satisfied, or to produce an error diagnostic if it is not.

Instance Declarations: The instances of any given class are described by a collection of instance declarations. For example, the following declarations show how one might define equality for booleans, and for pairs:

```
instance Eq Bool where
  x == y = if x then y else not y

instance (Eq a, Eq b) ⇒ Eq (a, b) where
  (x, y) == (u, v) = (x == u && y == v)
```

The first line of the second instance declaration tells us that an equality on values of types a and b is needed to provide an equality on pairs of type (a, b). No such preconditions are need for the definition of equality on booleans. Even with just these two declarations, we have already specified an equality operation on the infinite family of types that can be constructed from *Bool* by repeated uses of pairing. Additional declarations, which may be distributed over many modules, can be used to extend the class to include other datatypes.

3 Example: Building a Library of Collection Types

One of the most commonly suggested applications for multiple parameter type classes is to provide uniform interfaces to a wide range of collection types [10].

Such types might be expected to offer ways to construct empty collections, to insert values, to test for membership, and so on. The following declaration, greatly simplified for the purposes of presentation, introduces a two parameter class *Collects* that could be used as the starting point for such a project:

```
class Collects e ce where
   empty  :: ce
   insert :: e → ce → ce
   member :: e → ce → Bool
```

The type variable e used here represents the element type, while ce is the type of the collection itself. Within this framework, we might want to define instances of this class for lists or characteristic functions (both of which can be used to represent collections of any equality type), bit sets (which can be used to represent collections of characters), or hash tables (which can be used to represent any collection whose elements have a hash function). Omitting standard implementation details, this would lead to the following declarations:

```
instance Eq e ⇒ Collects e [e] where ...
instance Eq e ⇒ Collects e (e → Bool) where ...
instance Collects Char BitSet where ...
instance (Hashable e, Collects e ce)
                    ⇒ Collects e (Array Int ce) where ...
```

All this looks quite promising; we have a class and a range of interesting implementations. Unfortunately, there are some serious problems with the class declaration. First, the *empty* function has an ambiguous type:

```
empty :: Collects e ce ⇒ ce.
```

By ‘ambiguous’ we mean that there is a type variable e that appears on the left of the ⇒ symbol, but not on the right. The problem with this is that, according to the theoretical foundations of Haskell overloading, we cannot guarantee a well-defined semantics for any term with an ambiguous type [2, 4]. For this reason, a Haskell system will reject any attempt to define or use such terms.

We can sidestep this specific problem by removing the *empty* member from the class declaration. However, although the remaining members, *insert* and *member*, do not have ambiguous types, we still run into problems when we try to use them. For example, consider the following two functions:

```
f x y coll = insert x (insert y coll)
g coll     = f True 'a' coll
```

for which Hugs infers the following types:

```
f :: (Collects a c, Collects b c) ⇒ a → b → c → c
g :: (Collects Bool c, Collects Char c) ⇒ c → c.
```

Notice that the type for f allows the parameters x and y to be assigned different types, even though it attempts to insert each of the two values, one after the other, into the same collection, *coll*. If we hope to model collections that contain only one type of value, then this is clearly an inaccurate type. Worse still, the definition for g is accepted, without causing a type error. Thus the error in this code will not be detected at the point of definition, but only at the point of use, which might not even be in the same module. Obviously, we would prefer to avoid these problems, eliminating ambiguities, inferring more accurate types, and providing earlier detection of type errors.

3.1 An Attempt to Use Constructor Classes

Faced with the problems described above, some Haskell programmers might be tempted to use something like the following version of the class declaration:

```
class Collects e c where
  empty  :: c e
  insert :: e → c e → c e
  member :: e → c e → Bool
```

In fact this is precisely the approach taken by Okasaki [9], and by Peyton Jones [10], in more realistic attempts to build this kind of library. The key difference here is that we abstract over the type constructor c that is used to form the collection type $c\ e$, and not over that collection type itself, represented by *ce* in the original class declaration. Thus *Collects* is an example of a *constructor class* [6] in which the second parameter is a unary type *constructor*, replacing the nullary type parameter *ce* that was used in the original definition. This change avoids the immediate problems that we mentioned above:

- The *empty* operator has type $\mathit{Collects}\ e\ c \Rightarrow c\ e$, which is not ambiguous because both e and c appear on the right of the ⇒ symbol.
- The function f is assigned a more accurate type:

$$f :: (\mathit{Collects}\ e\ c) \Rightarrow e \to e \to c\ e \to c\ e.$$

- The function g is now rejected, as required, with a type error because the type of f does not allow the two arguments to have different types.

This, then, is an example of a multiple parameter class that does actually work quite well in practice, without ambiguity problems. The reason that it works, at least intuitively, is that its two parameters are essentially independent of one another and so there is a good fit with the interpretation of *Collects* as a relatively unconstrained relation between types e and type constructors c.

Unfortunately, this version of the *Collects* class is not as general as the original class seemed to be. Only one of the four instances listed in Section 3 can be used with this version of *Collects* because only one of them—the instance for lists—has a collection type that can be written in the form $c\ e$, for some type constructor c, and element type e. Some of the remaining instances can be reworked to fit the constructor class framework by introducing dummy type and value constructors, as in the following example:

```
newtype CharFun e = MkCharFun (e → Bool)
instance Eq e ⇒ Collects e CharFun where ...
```

This approach, however, is not particularly attractive. It clutters up programs with the artificial type constructor *CharFun*, and with uses of the value constructor *MkCharFun* to convert between the two distinct but equivalent representations of characteristic functions. The workaround is also limited, and cannot, in general, deal with cases like the *BitSet* example, where the element type is fixed and not a variable e that we can abstract over.

3.2 Using Parametric Type Classes

Another alternative is to use *parametric type classes* [3] (PTC), with predicates of the form $ce \in Collects\ e$, meaning that ce is a member of the class $Collects\ e$. Intuitively, there is one type class $Collects\ e$ for each choice of the e parameter. The definition of a parametric *Collects* class looks much like the original:

```
class ce ∈ Collects e where
  empty  :: ce
  insert :: e → ce → ce
  member :: e → ce → Bool
```

All of the instances declarations that we gave for the original *Collects* class in Section 3 can be adapted to the syntax of PTC, without introducing artificial type constructors. What makes it different from the two parameter class in Section 3 is the implied assumption that the element type e is uniquely determined by the collection type ce. A compiler that supports PTC must ensure that the declared instances of *Collects* do not violate this property. In return, it can use this information to avoid ambiguity and to infer more accurate types. For example, the type of *empty* is now $\forall e, ce.(ce \in Collects\ e) \Rightarrow ce$, and we do not need to treat this as being ambiguous because the unknown element type e is uniquely determined by ce.

Thus, PTC provides exactly the tools that we need to define and work with a library of collection classes. In our opinion, the original work on PTC has not received the attention that it deserves. In part, this may be because it was seen, incorrectly, as an alternative to constructor classes and not, more accurately, as an orthogonal extension. In addition, there has never been even a prototype implementation for potential users to experiment with.

3.3 Using Functional Dependencies

In this paper, we describe a generalization of parametric type classes that allows programmers to declare explicit *functional dependencies* between the parameters of a predicate. For example, we can achieve the same effects as PTC, with no further changes in notation, by annotating the original class definition with a dependency $ce \leadsto e$, to be read as "ce uniquely determines e."

class *Collects e ce* | $ce \leadsto e$ **where**
  empty :: *ce*
  insert :: $e \to ce \to ce$
  member :: $e \to ce \to Bool$

More generally, we allow class declarations to be annotated with (zero or more) dependencies of the form $(x_1, \ldots, x_n) \leadsto (y_1, \ldots, y_m)$, where $x_1, \ldots, x_n$, and $y_1, \ldots, y_m$ are type variables and $m, n > 0$[1]. Such a dependency is interpreted as an assertion that the y parameters are uniquely determined by the x parameters. Dependencies appear only in class declarations, and not in any other part of the language: the syntax for instance declarations, class constraints, and types is completely unchanged. For convenience, we allow the parentheses around a list of type variables in a dependency to be omitted if only a single variable is used.

This approach is strictly more general than PTC because it allows us to express a larger class of dependencies, including mutual dependencies such as $\{a \leadsto b,\ b \leadsto a\}$. It is also easier to integrate with the existing syntax of Haskell because it does not require any changes to the syntax of predicates.

By including dependency information, programmers can specify multiple parameter classes more precisely. To illustrate this, consider the following examples:

class *C a b* **where** ...
class *D a b* | $a \leadsto b$ **where** ...
class *E a b* | $a \leadsto b,\ b \leadsto a$ **where** ...

From the first declaration, we can tell only that C is a binary relation. The dependency $a \leadsto b$ in the second declaration tells us that D is not just a relation, but actually a (partial) function. From the two dependencies in the last declaration, we can see that E represents a (partial) one-one mapping.

The compiler is responsible for ensuring that the instances in scope at any given point are consistent with any declared dependencies[2]. For example, the following declarations cannot appear together because they violate the dependency for D, even though either one on its own would be acceptable:

instance *D Bool Int* **where** ...
instance *D Bool Char* **where** ...

Note also that the following declaration is not allowed, even by itself:

instance *D* [*a*] *b* **where** ...

The problem here is that this instance would allow one particular choice of $[a]$ to be associated with more than one choice for b, which contradicts the dependency specified in the definition of D. More generally, this means that, in any

[1] For practical reasons, a slightly different syntax is used for dependencies in the current prototype implementation, details of which are included in the distribution.

[2] Superclass declarations are handled in a similar way, leaving the compiler to ensure that every instance of a given class is also an instance of any superclasses.

declaration of the form **instance** $\ldots \Rightarrow D\ t\ s$ **where** $\ldots$, for some particular types t and s, the only variables that can appear in s are the ones that appear in t, and hence, if the type t is known, then s will be uniquely determined.

4 Further Examples

This section presents two additional examples to show how the use of functional dependencies can allow us to give more accurate specifications and to make more practical use of multiple parameter type classes.

Arithmetic Operations The Haskell prelude treats arithmetic functions like addition $(+)$ and multiplication $(*)$ as functions of type $\mathit{Num}\ a \Rightarrow a \to a \to a$, which means that the result will always be of the same type as the arguments. A more flexible approach would allow different argument types so that we could add two *Int* values to get an *Int* result, or add an *Int* to a *Float* to get a *Float* result. This more flexible approach can be coded as follows:

$$\begin{array}{l}
\textbf{class}\ \mathit{Add}\ a\ b\ c \mid (a,\ b) \leadsto c\ \textbf{where}\ (+) :: a \to b \to c \\
\textbf{class}\ \mathit{Mul}\ a\ b\ c \mid (a,\ b) \leadsto c\ \textbf{where}\ (*) :: a \to b \to c
\end{array}$$

$$\begin{array}{l}
\textbf{instance}\ \mathit{Mul}\ \mathit{Int}\ \mathit{Int}\ \mathit{Int}\ \textbf{where}\ \ldots \\
\textbf{instance}\ \mathit{Mul}\ \mathit{Int}\ \mathit{Float}\ \mathit{Float}\ \textbf{where}\ \ldots \\
\textbf{instance}\ \mathit{Mul}\ \mathit{Float}\ \mathit{Int}\ \mathit{Float}\ \textbf{where}\ \ldots \\
\textbf{instance}\ \mathit{Mul}\ \mathit{Float}\ \mathit{Float}\ \mathit{Float}\ \textbf{where}\ \ldots
\end{array}$$

In a separate linear algebra package, we might further extend our classes with arithmetic operations on vectors and matrices:

$$\begin{array}{l}
\textbf{instance}\ \mathit{Mul}\ a\ b\ c \Rightarrow \mathit{Mul}\ a\ (\mathit{Vec}\ b)\ (\mathit{Vec}\ c)\ \textbf{where}\ \ldots \\
\textbf{instance}\ \mathit{Mul}\ a\ b\ c \Rightarrow \mathit{Mul}\ a\ (\mathit{Mat}\ b)\ (\mathit{Mat}\ c)\ \textbf{where}\ \ldots \\
\textbf{instance}\ (\mathit{Mul}\ a\ b\ c,\ \mathit{Add}\ c\ c\ d) \\
\qquad\qquad \Rightarrow \mathit{Mul}\ (\mathit{Mat}\ a)\ (\mathit{Mat}\ b)\ (\mathit{Mat}\ d)\ \textbf{where}\ \ldots
\end{array}$$

Without dependency information, we quickly run into problems with ambiguity. For example, even simple expressions like $(1 * 2) * 3$ have ambiguous types:

$$(1 * 2) * 3 :: (\mathit{Mul}\ \mathit{Int}\ \mathit{Int}\ a,\ \mathit{Mul}\ a\ \mathit{Int}\ b) \Rightarrow b.$$

Using the dependencies, however, we can determine that $a = \mathit{Int}$, and then that $b = \mathit{Int}$, and so deduce that the expression has type *Int*. This example shows that it can be useful to allow multiple types on the left hand side of a dependency.

Finite Maps A *finite map* is an indexed collection of elements that provides operations to lookup the value associated with a particular index, or to add a new binding. This can be described by a class:

$$\begin{array}{l}
\textbf{class}\ \mathit{FiniteMap}\ i\ e\ \mathit{fm} \mid \mathit{fm} \leadsto (i,\ e)\ \textbf{where} \\
\quad \begin{array}{ll}
\mathit{emptyFM} & :: \mathit{fm} \\
\mathit{lookup} & :: i \to \mathit{fm} \to \mathit{Maybe}\ e \\
\mathit{extend} & :: i \to e \to \mathit{fm} \to \mathit{fm}
\end{array}
\end{array}$$

Here, *fm* is the finite map type, which uniquely determines both the index type i and the element type e. Association lists, functions, and arrays all fit naturally into this framework. We can also use a bit set as an indexed collection of booleans:

```
instance (Eq i) ⇒ FiniteMap i e [(i, e)] where ...
instance (Eq i) ⇒ FiniteMap i e (i → e) where ...
instance (Ix i) ⇒ FiniteMap i e (Array i e) where ...
instance FiniteMap Int Bool BitSet where ...
```

This is a variation on the treatment of collection types in Section 3, and, if the dependency is omitted, then we quickly run into very similar kinds of problem. We have included this example here to show that it can be useful to allow multiple types on the right hand side of a dependency.

5 Relations and Functional Dependencies

In this section, we provide a brief primer on the theory of relations and functional dependencies, as well as a summary of our notation. These ideas were originally developed as a foundation for relational database design [1]. They are well-established, and more detailed presentations of the theory, and of useful algorithms for working with them in practical settings, can be found in standard textbooks on the theory of databases [8]. A novelty of the current paper is in applying them to the design of a type system.

5.1 Relations

Following standard terminology, a *relation* R over an indexed family of sets $\{D_i\}_{i \in I}$ is just a set of *tuples*, each of which is an indexed family of values $\{t_i\}_{i \in I}$ such that $t_i \in D_i$ for each $i \in I$. More formally, R is just a subset of $\Pi i \in I.D_i$, where a tuple $t \in (\Pi i \in I.D_i)$ is a function that maps each index value $i \in I$ to a value $t_i \in D_i$ called the ith component of t. In the special case where $I = \{1, \ldots, n\}$, this reduces to the familiar special case where tuples are values $(t_1, \ldots, t_n) \in D_1 \times \ldots \times D_n$. If $X \subseteq I$, then we write t_X, pronounced "t at X", for the restriction of a tuple t to X. Intuitively, t_X just picks out the values of t for the indices appearing in X, and discards any remaining components.

5.2 Functional Dependencies

In the context of an index set I, a *functional dependency* is a term of the form $X \leadsto Y$, read as "X determines Y," where X and Y are both subsets of I. If a relation satisfies a functional dependency $X \leadsto Y$, then the values of any tuple at Y are uniquely determined by the values of that tuple at X. For example, taking $I = \{1, 2\}$, relations satisfying $\{\{1\} \leadsto \{2\}\}$ are just partial functions from D_1 to D_2, while relations satisfying $\{\{1\} \leadsto \{2\}, \{2\} \leadsto \{1\}\}$ are partial, injective functions from D_1 to D_2.

If F is a set of functional dependencies, and $J \subseteq I$ is a set of indices, then the *closure* of J with respect to F, written J^+_F is the smallest set such that $J \subseteq J^+_F$, and that, if $(X \leadsto Y) \in F$, and $X \subseteq J^+_F$, then $Y \subseteq J^+_F$. For example, if $I = \{1, 2\}$, and $F = \{\{1\} \leadsto \{2\}\}$, then $\{1\}^+_F = I$, and $\{2\}^+_F = \{2\}$. Intuitively, the closure J^+_F is the set of indices that are uniquely determined, either directly or indirectly, by the indices in J and the dependencies in F. Closures like this are easy to compute using a simple fixed point iteration.

6 Typing with Functional Dependencies

This section explains how to extend an implementation of Haskell to deal with functional dependencies. In fact the tools that we need are obtained as a special case of improvement for qualified types [5]. We will describe this briefly here; space restrictions prevent a more detailed overview. To simplify the presentation, we will assume that there is a set of indices (i.e., parameter names), written I_C, and a corresponding set of functional dependencies, written F_C, for each class name C. We will also assume that all predicates are written in the form $C\ t$, where t is a tuple of types indexed by I_C. This allows us to abstract away from the order in which the components are written in a particular implementation.

The type system of Haskell can be described using judgements of the form $P \mid A \vdash E : \tau$. Each such judgement represents an assertion that an expression E can be assigned a type τ, using the assumptions in A to type any free variables, and providing that the predicates in P are satisfied. When we say that a set of predicates is satisfied, we mean that they are all implied by the class and instance declarations that are in scope at the corresponding point in the program. For a given A and E, the goal of type inference is to find the most general choices for P and τ such that $P \mid A \vdash E : \tau$. If successful, we can infer a principal type for E by forming the qualified type $P \Rightarrow \tau$—without looking at the predicates in P—and then quantifying over all variables that appear in $P \Rightarrow \tau$ but not in A.

One of the main results of the theory of improvement is that we can apply *improving substitutions* to the predicate set P at any point during type inference (and as often as we like), without compromising on a useful notion of principal types. Intuitively, an improving substitution is just a substitution that can be applied to a particular set of predicates without changing its satisfiability properties. To make this more precise, we will write $\lfloor P \rfloor$ for the set of satisfiable instances of P, which is defined by:

$$\lfloor P \rfloor = \{ SP \mid S \text{ is a substitution and the predicates in } SP \text{ are satisfied} \}.$$

In this setting, we say that S is an improving substition for P if $\lfloor P \rfloor = \lfloor SP \rfloor$, and if the only variables involved in S that do not also appear in P are 'new' or 'fresh' type variables. From a practical perspective, this simply means that the subsitution will not change the set of environments or the set of types at which a given value can be used. The restriction to new variables is necessary to avoid conflicts with other type variables that might already be in use.

Improvement cannot play a useful role in a standard Haskell type system: The language does not restrict the choice of instances for any given type class, and hence the only improving substitions that we can obtain are equivalent to an identity substitution. With the introduction of functional dependencies, however, we do restrict the set of instances that can be defined, and this leads to opportunities for improvment. For example, by prohibiting the definition of instances of the form $Collects\ a\ [b]$ where $a \neq b$, we know that we can use an improving substitution $[a/b]$ and map any such predicate into the form $Collects\ a\ [a]$.

6.1 Ensuring that Dependencies are Valid

Our first task is to ensure that all declared instances for a class C are consistent with the functional dependencies in F_C. For example, suppose that we have an instance declaration for C of the form:

$$\textbf{instance}\ \ldots \Rightarrow C\ t\ \textbf{where}\ \ldots$$

Now, for each $(X \leadsto Y) \in F_C$, we must ensure that $TV(t_Y) \subseteq TV(t_X)$ or otherwise the elements of t_Y might not be uniquely determined by the elements of t_X. (The notation $TV(X)$ refers to the set of type variables appearing free in the object X.) A further restriction is needed to ensure pairwise compatibility between instance declarations for C. For example, if we have a second instance:

$$\textbf{instance}\ \ldots \Rightarrow C\ s\ \textbf{where}\ \ldots,$$

and a dependency $(X \leadsto Y) \in F_C$, then we must ensure that $t_Y = s_Y$ whenever $t_X = s_X$. In fact, on the assumption that the two instances will normally contain type variables—which could later be instantiated to more specific types—we will actually need to check that: for all (kind-preserving) substitutions S, if $St_X = Ss_X$, then $St_Y = Ss_Y$. It is easy to see that this test can be reduced to checking that, if t_X and s_X have a most general unifier U, then $Ut_Y = Us_Y$. This is enough to guarantee that the declared dependencies are satisfied. For example, the instance declarations in Section 3 are consistent with the dependency $ce \leadsto e$.

6.2 Improving Inferred Types

There are two ways that a dependency $(X \leadsto Y) \in F_C$ for a class C can be used to help infer more accurate types:

- If we have predicates $(C\ t)$ and $(C\ s)$ with $t_X = s_X$, then t_Y and s_Y must be equal.
- Suppose that we have an inferred predicate $C\ t$, and an instance:

  $$\textbf{instance}\ \ldots \Rightarrow C\ t'\ \textbf{where}\ \ldots$$

  If $t_X = St'_X$, for some substitution S (which could be calculated by one-way matching), then t_Y and St'_Y must be equal.

In both cases, we can use unification to ensure that the equalities are satisfied, and to calculate a suitable improving substitution [5]. If unification fails, then we have detected a type error. Note that we will, in general, need to iterate this process until no further opportunities for improvement can be found.

6.3 Detecting Ambiguity

As mentioned in Section 3, we cannot guarantee a well-defined semantics for any function that has an ambiguous type. With the standard definition, a type of the form $(\forall a_1 \ldots . \forall a_n . P \Rightarrow \tau)$ is ambiguous if $(\{a_1, \ldots, a_n\} \cap TV(P)) \not\subseteq TV(\tau)$, indicating that one of the quantified variables a_i appears in $TV(P)$ but not in $TV(\tau)$. Our intuition is that, if there is no reference to a_i in the body of the type, then there will be no way to determine how it should be bound when the type is instantiated. However, in the presence of functional dependencies, there might be another way to find the required instantiation of a_i. We need not insist that every $a \in TV(P)$ is mentioned explicitly in τ, so long as they are all uniquely determined by the variables in $TV(\tau)$.

The first step to formalizing this idea is to note that every set of predicates P induces a set of functional dependencies F_P on the type variables in $TV(P)$:

$$F_P = \{\, TV(t_X) \leadsto TV(t_Y) \mid (C\ t) \in P,\ (X \leadsto Y) \in F_C \,\}.$$

This has a fairly straightforward reading: if all of the variables in t_X are known, and if $X \leadsto Y$, then the components of t at X are also known, and hence so are the components, and thus the type variables, in t at Y.

To determine if a type $(\forall a_1 \ldots . \forall a_n . P \Rightarrow \tau)$ is ambiguous, we calculate the set of dependencies F_P, and then take the closure of $TV(\tau)$ with respect to F_P to obtain the set of variables that are determined by τ. The type is ambiguous only if there are variables a_i in P that are not included in this closure. More concisely, the type is *ambiguous* if, and only if $(\{a_1, \ldots, a_n\} \cap TV(P)) \not\subseteq (TV(\tau))^+_{F_P}$.

On a related point, we note that current implementations of Haskell are required to check that, in any declaration of the form **instance** $P \Rightarrow C\ t$ **where** ..., only the variables appearing in t can be used in P (i.e., we must ensure that $TV(P) \subseteq TV(t)$). In light of the observations that have been made in this section, we can relax this to require only that $TV(P) \subseteq (TV(t))^+_{F_P}$. Thus P may contain variables that are not explicitly mentioned in t, provided that they are still determined by the variables in t.

6.4 Generalizing Inferred Types

In a standard Hindley-Milner type system, principal types are computed using a process of generalization. Given an inferred but unquantified type $P \Rightarrow \tau$, we would normally just calculate the set of type variables $T = TV(P \Rightarrow \tau)$, over which we might want to quantify, and the set of variables $V = TV(A)$ that are fixed in the current assumptions A, and then quantify over any variables in the difference, $T \setminus V$. In the presence of functional dependencies, however, we must be a little more careful: a variable a that appears in T but not in V may still need to be treated as a fixed variable if it is determined by V. To account for this, we should only quantify over the variables in $T \setminus V^+_{F_P}$.

7 Putting a Name to Functional Dependencies

The approach described in this paper provides a way for programmers to indicate that there are dependencies between the parameters of a type class, but stops short of giving those dependencies a name. To illustrate this point, consider the following pair of class declarations:

$$\begin{array}{l}\textbf{class}\ U\ \ a\ \ b \mid a \rightsquigarrow b\ \textbf{where}\ \ldots\\ \textbf{class}\ U\ \ a\ \ b \Rightarrow V\ \ a\ \ b\ \textbf{where}\ \ldots\end{array}$$

From the first declaration, we know that there is a dependency between the parameters of U; should there not also be a dependency between the parameters of V, inherited from its superclass U? Such a dependency could be added by changing the second declaration to:

$$\textbf{class}\ U\ \ a\ \ b \Rightarrow V\ \ a\ \ b \mid a \rightsquigarrow b\ \textbf{where}\ \ldots$$

but this tells only part of the story. For example, given two predicates $U\ \ a\ \ b$ and $V\ \ a\ \ c$, nothing in the rules from Section 6 will allow us to infer that $b = c$. Let us return to the dependency on U and give a name to it by writing u for the function that maps each a to the b that it determines. This might even be made explicit in the syntax of the language by changing the declaration to read:

$$\textbf{class}\ U\ \ a\ \ b \mid u :: a \rightsquigarrow b\ \textbf{where}\ \ldots$$

Now we can change the declaration of V again to indicate that it inherits the same dependency u:

$$\textbf{class}\ U\ \ a\ \ b \Rightarrow V\ \ a\ \ b \mid u :: a \rightsquigarrow b\ \textbf{where}\ \ldots$$

Now, given the predicates $U\ \ a\ \ b$ and $V\ \ a\ \ c$, we can infer that $b = u\ \ a = c$, as expected. It is not yet clear how useful this particular feature might be, or whether it might be better to leave the type checker to infer inherited dependencies automatically, without requiring the programmer to provide names for them. The current prototype includes an experimental implementation of this idea (without making dependency names explicit), but the interactions with other language features, particularly overlapping instances, are not yet fully understood. Careful exploration of these issues is therefore a topic for future work. However, the example does show that there are further opportunities to exploit dependency information that go beyond the ideas described in Section 6.

8 Conclusions and Future Work

The ideas described in this paper have been implemented in the latest version of the Hugs interpreter [7], and seem to work well in practice. Pleasingly, some early users have already found new applications for this extension in their own work, allowing them to overcome problems that they had previously been unable to fix. Others have provided feedback that enabled us to discover places where further use of dependency information might be used, as described in Section 7.

In constructing this system, we have used ideas from the theory of relational databases. One further interesting area for future work would be to see if other ideas developed there could also be exploited in the design of programming language type systems. Users of functional languages are, of course, accustomed to working with parameterized datatypes. Functional dependencies provide a way to express similar relationships between types, without being quite so specific. For example, perhaps similar ideas could be used in conjunction with existential types to capture dependencies between types whose identities have been hidden?

Acknowledgments

I would like to thank my colleagues at OGI for their interest in this work. Particular thanks go to Jeff Lewis for both insight and patches, and to Lois Delcambre for explaining the role that functional dependencies play in database theory.

References

[1] W. W. Armstrong. Dependency structures of data base relationships. In *IFIP Cong.*, Geneva, Switzerland, 1974.

[2] S. M. Blott. *An approach to overloading with polymorphism.* PhD thesis, Department of Computing Science, University of Glasgow, September 1991.

[3] K. Chen, P. Hudak, and M. Odersky. Parametric type classes (extended abstract). In *ACM conference on LISP and Functional Programming*, San Francisco, CA, June 1992.

[4] M. P. Jones. *Qualified Types: Theory and Practice.* PhD thesis, Programming Research Group, Oxford University Computing Laboratory, July 1992. Published by Cambridge University Press, November 1994.

[5] M. P. Jones. Simplifying and improving qualified types. In *International Conference on Functional Programming Languages and Computer Architecture*, pages 160–169, June 1995.

[6] M. P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1), January 1995.

[7] M. P. Jones and J. C. Peterson. *Hugs 98 User Manual*, September 1999.

[8] D. Maier. *The Theory of Relational Databases.* Computer Science Press, 1983.

[9] C. Okasaki. *Edison User's Guide*, May 1999.

[10] S. Peyton Jones. Bulk types with class. In *Proceedings of the Second Haskell Workshop*, Amsterdam, June 1997.

[11] S. Peyton Jones and J. Hughes, editors. *Report on the Programming Language Haskell 98, A Non-strict Purely Functional Language*, February 1999.

[12] S. Peyton Jones and J. Hughes (editors). *Standard libraries for the Haskell 98 programming language*, February 1999.

[13] S. Peyton Jones, M. Jones, and E. Meijer. Type classes: Exploring the design space. In *Proceedings of the Second Haskell Workshop*, Amsterdam, June 1997.

[14] P. Wadler and S. Blott. How to make *ad hoc* polymorphism less *ad hoc.* In *Proceedings of 16th ACM Symposium on Principles of Programming Languages*, pages 60–76, Jan 1989.

Sharing Continuations: Proofnets for Languages with Explicit Control

Julia L. Lawall[*] and Harry G. Mairson[**]

Department of Computer Science
111 Cummington Street
Boston University
Boston, Massachusetts 02215
{jll,mairson}@cs.bu.edu

Abstract. We introduce graph reduction technology that implements functional languages with control, such as Scheme with `call/cc`, where continuations can be manipulated explicitly as values, and can be optimally reduced in the sense of Lévy. The technology is founded on *proofnets* for multiplicative-exponential linear logic, extending the techniques originally proposed by Lamping, where we adapt the continuation-passing style transformation to yield a new understanding of sharable values. Confluence is maintained by returning multiple answers to a (shared) continuation.

Proofnets provide a *concurrent* version of linear logic proofs, eliminating structurally irrelevant sequentialization, and ignoring asymmetric distinctions between inputs and outputs—dually, expressions and continuations. While Lamping's graphs and their variants encode an embedding of intuitionistic logic into linear logic, our construction implicitly contains an embedding of classical logic into linear logic.

We propose a family of translations, produced uniformly by beginning with a continuation-passing style semantics for the languages, employing standard codings into proofnets using call-by-value, call-by-name—or hybrids of the two—to locate proofnet *boxes*, and converting the proofnets to direct style. The resulting graphs can be reduced simply (cut elimination for linear logic), have a consistent semantics that is preserved by reduction (geometry of interaction, via the so-called *context semantics*), and allow shared, incremental evaluation of continuations (optimal reduction).

1 Introduction

Expressions and continuations are dual, separate but equal computational structures in a programming language. The former provides a value; the latter consumes it. Yet evaluating expressions is very familiar, while evaluating continuations

[*] Supported by NSF Grant EIA-9806718.

[**] Supported by NSF Grants CCR-9619638 and EIA-9806718, and the Tyson Foundation.

G. Smolka (Ed.): ESOP/ETAPS 2000, LNCS 1782, pp. 245–259, 2000.

is considered esoteric, even though both are made of the same stuff. The incorporation of continuations as first-class citizens in programming languages was not welcomed like the Emancipation Proclamation, but instead regarded warily as a kind of witchcraft, with implementation pragmatics that are ill-defined and unclear. If expressions and continuations are indeed dual, then so should be the technology of their implementation, and the flexibility with which we reason about them. Efficient evaluation of one should reveal dual strategies for evaluating the other. In short, everything we know about expressions we ought to know about continuations.

We take a significant step towards this equality by formulating a general version of graph reduction that implements the sharing and optimal incremental evaluation of both expressions and continuations, each evaluated using the same primitive operations. By founding our technology on generic tools from logic and programming language theory, specifically the CPS transform and its relation to linear logic, we are for the first time able to produce a family of related implementations in an entirely mechanical way.

Nishizaki earlier produced a coding of Scheme with `call/cc` in linear logic, via *ad hoc* reasoning, based on a proof of a proposition of linear logic corresponding to the type of `call/cc` [22]. In contrast, our new contribution is to produce Nishizaki's coding, and many others, by a mechanical process based on the denotational semantics of the programming language. Not only do we get a much deeper insight into principles, we greatly simplify the problem of constructing graph reduction implementations of other languages with explicit control. In bringing ideas from logical theory closer to implementation technology, we hope to make researchers think about the pragmatics of continuations in simple, novel, and useful ways.

Our methodology is founded on proofnets for multiplicative-exponential linear logic, following the beautiful insights of John Lamping [17], who realized Jean-Jacques Lévy's specification of correct, optimal reduction for the λ-calculus [18], and of Gonthier, Abadi, and Lévy, who reinterpreted Lamping's insights in the guise of Girard's geometry of interaction, and the related embedding of intuitionistic logic in linear logic [12,13]. Linear logic [11] provides an ideal substrate for the implementation of control operators, as it makes no asymmetric distinctions between inputs and outputs, or analogous expressions and continuations. We extend the optimal reduction technology to implement explicit control and sharing of continuations, essentially via an embedding of *classical* logic in linear logic, following a line of research beginning with Griffin and then Murthy [14,21]. Our construction is based on continuation-passing style, but generates direct-style graphs. This approach extends to implement most any functional language with abortive control operators whose semantics can be described in continuation-passing style—for example, Filinski's *symmetric* λ*-calculus* [8], and Parigot's λ_μ*-calculus* [23].[1]

[1] We have implemented these languages using the techniques in this paper. This work will be included in a later, extended version of the manuscript.

What do optimality and correctness mean in a language with explicit control? To understand *optimality* and *sharing* in the context of continuations, consider the evaluation of a Scheme expression

```
([fun1]
   (call/cc (lambda (a) ([fun2]
      (call/cc (lambda (b) [exp]))))))
```

in the context of some complex continuation k. The expression [*exp*] can *implicitly* access its current continuation c, or *explicitly* access continuations a and b named by `a` and `b`. All of these continuations extend the continuation of the entire expression k. Optimality ensures that a, b, and c share k, that b and c share a, and so on. If continuations are shared, duplicate work can be avoided as continuations are simplified.

A reduction strategy ρ is *correct* if for any expression E, if there is some strategy σ that reduces E to a normal form, then ρ also reduces E to a normal form. In the absence of control operators, normal forms in the λ-calculus are unique, so all correct strategies produce the same normal form, if one exists. However, control operators destroy the uniqueness of normal forms: let E be the Scheme expression `(call/cc (lambda (k) ((lambda (x) 1) (k 2))))`; a call-by-name strategy reduces E to 1, while a call-by-value strategy reduces E to 2. A correct evaluation strategy cannot simply choose one of these answers. Define context C as `(if (=` $[-]$ `1) 0` $\perp$`)` and C' as `(if (=` $[-]$ `1)` $\perp$ `0)`. Since $C[E]$ evaluates to 0 under call-by-name and diverges under call-by-value, a correct evaluation of E must return 1. But since $C'[E]$ evaluates to 0 under call-by-value and diverges under call-by-name, a correct evaluation of E must return 2. Returning both 1 and 2 in the evaluation of E is *not* contradictory: it merely amounts to supplying both answers to a single *shared* continuation.

Technical contributions: The efficiency of optimal reduction is based on the incremental propagation of sharing nodes. Implementations of optimal reduction based on linear logic, as proposed by Gonthier, Abadi, and Lévy, and later by Asperti, use proofnet *boxes* to coordinate these interactions. Nevertheless, the boxing strategy only permits the sharing of values. To extend this technology to languages with control operators, the key technical question is: where do we put the boxes to allow the sharing of continuations?

Our solution exploits the *continuation-passing style (CPS) transformation.* If a language with control operators can be translated into the pure λ-calculus using a CPS transformation, we can use existing technology to construct the graph of the CPS transformed term. The CPS translation of a term is more verbose than the original term, and more expensive to reduce to a normal form. We show that for a language with abortive continuations, the graphs of CPS terms can be mechanically converted back to direct style, maintaining the boxing of continuations induced by the CPS term. The transformation "rotates" principal ports of boxes so that continuations can be copied. We prove that this transformation does not change the underlying denotational semantics of the terms, as defined by the geometry of interaction. This approach can be applied to any variant of the CPS transformation, and any strategy for coding pure λ-terms as proofnets. The result is a family of possible translations into graphs, and these graphs can be optimally reduced in the sense of Lévy's labelled terms.

Traditionally, compiler optimizations have addressed sharing of expressions. The technology presented here provides a new systematic basis on which to optimize the sharing of continuations.

In summary, all of the translations we outline possess a simple graph reduction on translated terms (cut elimination for linear logic), a consistent semantics that is preserved by reduction (geometry of interaction, via the so-called *context semantics* of Gonthier [12]), and a mechanism whereby continuations can be incrementally evaluated (optimal reduction). The situation of this technology within multiplicative-exponential linear logic ensures that the semantic characterization given is equivalent to the operational semantics of graph reduction. Viewing data types as games, and contexts (in the sense of Gonthier) as moves in a composite game, one immediately suspects that categories of games should provide the right kind of "more abstract" semantics for calculi with explicit control. Furthermore, full abstraction theorems for languages with control seem to be easily accessible, given the full completeness results for linear logic.

2 Preliminaries

We briefly sketch the construction of graphs to implement λ-calculus; more details can be found elsewhere [1,12]. Graphs are composed of wires and fixed-arity nodes, as well as *boxes*, which enclose subgraphs. The λ-calculus is encoded using apply nodes (@), lambda nodes (λ), sharing nodes (▽), weakening nodes (⊙), and croissants (⌒). A box allows a subgraph to be duplicated by a sharing node, or discarded by a weakening node. When sharing is no longer required, a croissant can open the box, allowing interaction with the subgraph inside. The meaning of the other nodes should be intuitive.

One port of each node or box is designated as the *principal port*. Other ports are *auxiliary ports*. Reduction takes place when two graph constructs are connected at their principal ports. A box can also interact with another box at its auxiliary port. Global reduction rules are shown in Figure 1, where black dots indicate principal ports. Graphs can also be reduced by local reduction rules, described elsewhere [1,12]. Local reduction of the graph of a λ-term implements Lévy's *optimal reduction* [18].

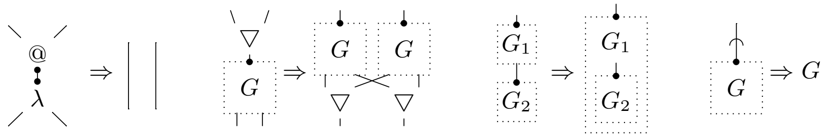


Fig. 1. Global reduction rules

The big question in implementing λ-calculus within this framework is where to put the boxes to allow the unrestricted sharing of values. We mention two commonly used boxing disciplines; see [19] for others. The *call-by-value (CBV) coding* boxes the graph of every λ-abstraction. Correspondingly, a croissant is placed on the function position of every apply node. The *call-by-name (CBN) coding* boxes the graph of the argument of every function application. Correspondingly, a variable reference is implemented by a croissant. These codings amount to Curry-Howard style embeddings of intuitionistic logic in linear logic. Figure 2 illustrates the CBN coding of the λ-calculus, which we will use in this paper. Note that the left side of a lambda node leads to the graph of the body, while the right side leads to the (perhaps shared) occurrence of the bound variable. Correspondingly, the left side of an apply node leads to the context of the application, while the right side leads to the argument. Our results are equally applicable to the CBV coding, and to any other consistent boxing strategy.

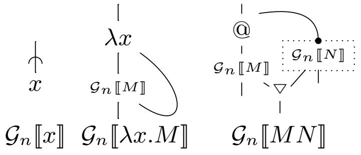


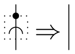
Fig. 2. CBN coding of λ-terms

Graphs of simply-typed λ-terms can be assigned linear-logic types. In particular, the type of a box is $!A$, allowing sharing of the A-typed value inside. Regardless of the boxing strategy, the constraints of linear logic typing imply:

Proposition 1. *Boxes never get in the way of β reductions.*

As a consequence, optimal (local) reduction reduces any two graphs with the same arrangement of apply, lambda, and sharing nodes in the same way.

Asperti has proposed some optimizations to these boxing strategies [1]. The simplest is to apply the following rule to the translation of a λ-term:



We apply this optimization, without comment, throughout the paper.

Implementing control operators: The above codings make the continuation and argument of an application equally accessible, the former on the left side, and the latter on the right side of the apply node. References to these values are similiarly equally accessible to a λ-abstraction, at the left and right side of the lambda node. Because the λ-calculus can only express the sharing of arguments, via parameter binding, the boxing strategies only ensure that the value of the argument is boxed. Control operators such as Scheme's `call/cc`, however, introduce the possibility to name, and thus duplicate and discard, the continuation.

3 Continuations in the λ-Calculus

We now derive a family of graph encodings for terms in the λ-calculus with `call/cc` from encodings of the corresponding CPS terms. For pure λ-terms, this approach produces graphs with the same arrangement of lambda, apply, and sharing nodes as previous translations, and thus such graphs reduce to normal form in the optimal number of beta steps, as given by Lévy's specification.

3.1 The CPS Transformation

Continuation-passing style (CPS) is a style of programming in which the continuation at each point is represented explicitly as a function. Because the continuation function makes explicit the remaining computation at the current program point, a CPS term necessarily specifies an evaluation order. A λ-calculus program can be converted to CPS automatically using a CPS transformation. Furthermore, the control operator `call/cc` can be translated into CPS. Typically a CPS transformation encodes a CBV or CBN evaluation order, however any consistent mixture is possible [15].

Plotkin's CBV and CBN CPS transformations, extended with the translation of `call/cc`, are shown in Figures 3 and 4, respectively [24]. For typed terms, these transformations induce a corresponding transformation on types. Define $\alpha^* \equiv \alpha$ for any base type α (including $\bot$) and $(\alpha \to \beta)^* \equiv \alpha^* \to \neg\neg\beta^*$ where $\neg\tau \equiv \tau \to \bot$; then the CBV CPS transformation maps a derivation $\{x : \sigma \in \Gamma\} \vdash E : \tau$ to $\{x : \sigma^* \mid x : \sigma \in \Gamma\} \vdash \mathcal{C}_v[\![E]\!] : \neg\neg\tau^*$. Similarly, define $\alpha^\dagger \equiv \alpha$, and $(\alpha \to \beta)^\dagger \equiv \neg\neg\alpha^\dagger \to \neg\neg\beta^\dagger$; then the CBN CPS transformation maps the same derivation to $\{x : \neg\neg\sigma^\dagger \mid x : \sigma \in \Gamma\} \vdash \mathcal{C}_n[\![E]\!] : \neg\neg\tau^\dagger$.

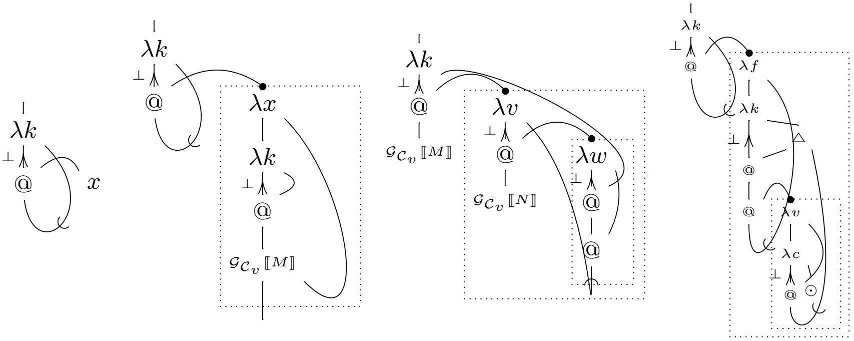
$$\begin{aligned}
\mathcal{C}_v[\![x]\!] &\equiv \lambda\kappa.\kappa x \\
\mathcal{C}_v[\![\lambda x.M]\!] &\equiv \lambda\kappa.\kappa(\lambda x.\lambda k.\mathcal{C}_v[\![M]\!]k) \\
\mathcal{C}_v[\![MN]\!] &\equiv \lambda\kappa.\mathcal{C}_v[\![M]\!](\lambda v.\mathcal{C}_v[\![N]\!](\lambda w.vw\kappa)) \\
\mathcal{C}_v[\![\texttt{call/cc}]\!] &\equiv \lambda\kappa.\kappa(\lambda f.\lambda k.f(\lambda v.\lambda c.kv)k)
\end{aligned}$$

Fig. 3. CBV CPS transformation of the λ-calculus, including `call/cc`

Replacing the λ-terms produced by a CPS transformation by the corresponding graphs gives a translation of terms into graphs in which the continuation is accessible as a sharable value. Figure 5 presents the graph translation corresponding to the CBV CPS transformations. We have used the CBN boxing strategy, although any strategy can be used. The translation corresponding to the CBN CPS transformation is similar. Only $\bot$ types are indicated.

$$\begin{aligned}
\mathcal{C}_n[\![x]\!] &\equiv \lambda\kappa.x\kappa \\
\mathcal{C}_n[\![\lambda x.M]\!] &\equiv \lambda\kappa.\kappa(\lambda x.\lambda k.\mathcal{C}_n[\![M]\!]k) \\
\mathcal{C}_n[\![MN]\!] &\equiv \lambda\kappa.\mathcal{C}_n[\![M]\!](\lambda v.v(\mathcal{C}_n[\![N]\!])\kappa) \\
\mathcal{C}_n[\![\texttt{call/cc}]\!] &\equiv \lambda\kappa.\kappa(\lambda f.\lambda k.f(\lambda a.a(\lambda q.q(\lambda v.\lambda c.vk))k))
\end{aligned}$$

Fig. 4. CBN CPS transformation of the λ-calculus, including `call/cc`


λk
⊥
@
x
λk
⊥
@
λx
λk
⊥
@
$\mathcal{G}_{\mathcal{C}_v}[\![M]\!]$
λk
⊥
@
$\mathcal{G}_{\mathcal{C}_v}[\![M]\!]$
λv
⊥
@
$\mathcal{G}_{\mathcal{C}_v}[\![N]\!]$
λw
⊥
@
@
λk
⊥
@
λf
λk
⊥
@
@
λv
λc
⊥
@
⊙

$\mathcal{G}_{\mathcal{C}_v}[\![x]\!]$ $\mathcal{G}_{\mathcal{C}_v}[\![\lambda x.M]\!]$ $\mathcal{G}_{\mathcal{C}_v}[\![MN]\!]$ $\mathcal{G}_{\mathcal{C}_v}[\![\texttt{call/cc}]\!]$

Fig. 5. Graph translation based on the CBV CPS transformation, and the CBN boxing strategy

This implementation strategy, while straightforward, is unsatisfactory. The CPS transformation introduces lambda and apply nodes that are not part of the original term. Thus, optimal reduction of the resulting graph does not reduce the original term using the minimal number of β steps. Indeed, the number of β steps is affected by the CPS transformation chosen. Furthermore, the graph translation does not exploit the symmetry between the left side of a lambda or apply node, which connects to the continuation of a function application, and the right side, which connects to the argument. The CPS encoding does, however, produce a graph in which continuations are consistently boxed. Thus, we would like a graph translation generating the same arrangement of lambda and apply nodes for pure λ-terms as the translations defined in Section 2, while retaining the boxing of continuations suggested by the CPS transformation.

3.2 The DS Transformation on Graphs

Essentially, we would like to eliminate the lambda and apply nodes that construct and manipulate continuations. To simplify the graph, we exploit the $\perp$ return type of every continuation and continuation abstraction. In a CPS program, we are not interested in the result of type $\perp$, but instead in the value passed to the initial continuation. We can show that the computation of a value of a non-$\perp$ type cannot depend on an edge transmitting a value of $\perp$ type. The simple conclusion is to remove all such edges. Because this transformation eliminates continuations from CPS graphs, we refer to it as the *direct-style (DS) transformation.*

Removing edges from the graph of course affects the nodes incident upon these edges. Figure 6 shows transformation rules sufficient to treat CPS terms. $\perp$-typed edges in other positions are treated similarly.

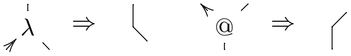


Fig. 6. DS graph transformation rules

While the graph codings of Section 2 were inspired by embeddings of intuitionistic logic into linear logic, they can equally well implement untyped terms. Here, however, we do require that $\perp$-typed values are used consistently.

The results of applying the DS transformation to the graph translations based on the CBV and CBN CPS transformations are shown in Figures 7 and 8, respectively. We refer to these translations as the $\text{CBV}_{\text{CPS/N}}$ and $\text{CBN}_{\text{CPS/N}}$ translations, respectively. Both achieve our goals: The arrangement of apply and lambda nodes in the translation of a pure λ-term is identical to that of the codings presented in Section 2, and continuations are boxed, allowing them to be duplicated or discarded.

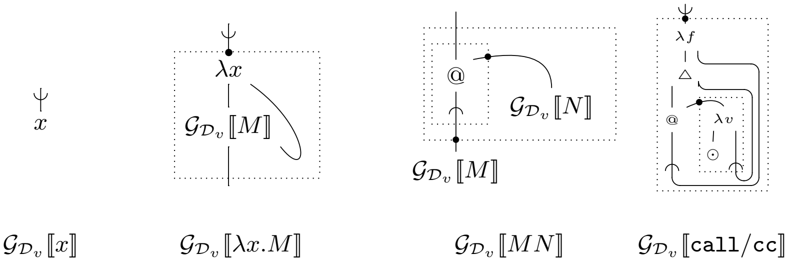
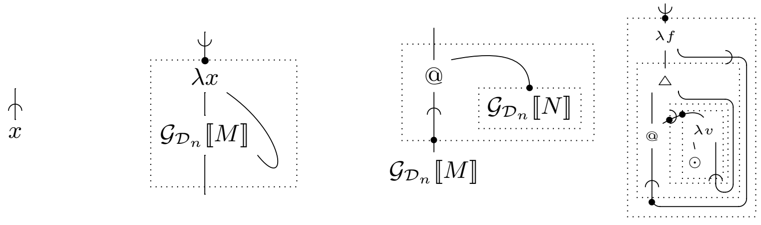

x
λx
$\mathcal{G}_{\mathcal{D}_v}[\![M]\!]$
@
$\mathcal{G}_{\mathcal{D}_v}[\![N]\!]$
$\mathcal{G}_{\mathcal{D}_v}[\![M]\!]$
λf
@
λv
$\mathcal{G}_{\mathcal{D}_v}[\![x]\!]$
$\mathcal{G}_{\mathcal{D}_v}[\![\lambda x.M]\!]$
$\mathcal{G}_{\mathcal{D}_v}[\![MN]\!]$
$\mathcal{G}_{\mathcal{D}_v}[\![\texttt{call/cc}]\!]$

Fig. 7. DS graph translation derived from the CBV CPS transformation

3.3 Correctness of the DS Transformation

A graph can be viewed as a set of apply and lambda nodes connected by edges that may contain sharing information, as controlled by sharing nodes, croissants, and box boundaries. Because the DS transformation only modifies (i.e., eliminates) apply and lambda nodes, it does not directly affect this sharing information.


x
λx
$\mathcal{G}_{\mathcal{D}_n}[\![M]\!]$
@
$\mathcal{G}_{\mathcal{D}_n}[\![N]\!]$
$\mathcal{G}_{\mathcal{D}_n}[\![M]\!]$
λf
@
λv

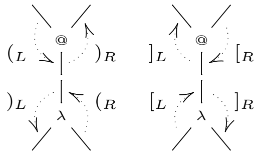
$\mathcal{G}_{\mathcal{D}_n}[\![x]\!]$ $\mathcal{G}_{\mathcal{D}_n}[\![\lambda x.M]\!]$ $\mathcal{G}_{\mathcal{D}_n}[\![MN]\!]$ $\mathcal{G}_{\mathcal{D}_n}[\![\texttt{call/cc}]\!]$

Fig. 8. DS graph translation derived from the CBN CPS transformation

Consider the path between two edges that are ultimately connected by reduction. We prove that the DS transformation maintains the way in which such a path traverses apply and lambda nodes. Because the arrangement of the other nodes is not modified by the DS transformation, they continue to behave in the same way as well.

Throughout, we assume the graph is simply typed.

Definition 1. *A* path *is a directed sequence of connected edges, labelled as shown below, where for each node traversed on the path, the two edges incident on the node connect respectively to the principal port, and to an auxiliary port. The label of a path is the concatenation of the labels of the edges.*



Definition 2. *A* well-balanced path *is a path whose label is described by the following grammar, where* 1 *is the empty word:*

$$B ::= 1 \mid (_L B)_L \mid (_R B)_R \mid [_L B]_L \mid [_R B]_R \mid BB$$

An unbalanced path *is a path that is not well-balanced and whose label is a subword of a label derivable from B.*

This definition of well-balanced path generalizes that of Asperti and Laneve [2] to include paths that cannot occur in the translation of an ordinary λ-term, but can occur in the image of the DS transformation. Note that some paths are neither well-balanced nor unbalanced, for example a path that enters an apply node on the left and immediately exits the next lambda node on the right, with label $(_L]_R$. Unbalanced paths describe correct information flow (in the sense of the geometry of interaction) in a well-typed graph, but cannot reduce to form a beta redex.

Proposition 2. *Let p be a path that is converted to an edge by the DS transformation. Then, p is either well-balanced or unbalanced.*

We can show the following by induction on the structure of a path in a well-typed graph:

Proposition 3. *In a well-balanced path, the first and last edges have identical types.*

An unbalanced path has a label of the form $B)X$ or $X(B$, where) and (are one of the four forms of open/closed parentheses, and X is a well-balanced or unbalanced path. Applying Proposition 3 to B, we can show:

Proposition 4. *In an unbalanced path, either the first or last edge has arrow type.*

For a path p, if $\mathcal{D}(p)$ is an edge, every node of p must be eliminated by the DS transformation. Thus, using Proposition 4, we can show:

Proposition 5. *Let p be an unbalanced path. If $\mathcal{D}(p)$ is an edge, then either the first or last edge of p has type $A \rightarrow \bot$, for some A.*

Theorem 1. (Soundness) *Let G be the graph of a pure λ-term. Then the diagram*

$$\begin{tikzcd}
G \arrow[r] \arrow[d, "\mathcal{D}"] & G' \arrow[d, "\mathcal{D}"] \\
\mathcal{D}(G) \arrow[r] & \mathcal{D}(G') \equiv G''
\end{tikzcd}$$

commutes, where:

1. *(The top path can be simulated by the bottom path): If G reduces to G' by global reductions, then $\mathcal{D}(G)$ reduces to $\mathcal{D}(G')$ by global reductions.*
2. *(The bottom path can be simulated by the top path): If $\mathcal{D}(G)$ reduces to G'' by global reductions, then there is some G' such that G reduces to G' by global reductions, and $G'' \equiv \mathcal{D}(G')$.*

Proof. To prove that the top path can be simulated by the bottom path, we use induction on the number of reduction steps in the top path. Observe that the effect of the DS transformation on a node is completely local, determined only by the types of edges incident on the node. Thus, the effect of the DS transformation on source nodes not affected by the reduction step is the same in both the source graph and the reduced graph; we need only consider the effect of the DS transformation on the nodes involved in the reduction step. Consider each possible redex in the source graph:

– A β-redex with function type $A \rightarrow B$, where $B \neq \bot$. Since the redex is not affected by the DS transformation, we can perform the same reduction step in $\mathcal{D}(G)$, and the resulting graph has the form $\mathcal{D}(G')$.

- A β-redex with function type $A \to \bot$. Reducing a beta redex creates an edge of the argument type (connecting the argument to the occurrence of the parameter) and an edge of the return type (connecting the result of the body to the context of the application). Thus, reducing a redex with function type $A \to \bot$ creates an edge of type A and an edge of type $\bot$ in G'. The edge of type $\bot$ is then eliminated by the DS transformation. Applying the DS transformation to G directly also eliminates the $\bot$-typed edges and connects the A-typed edges, thus having the same effect as beta reduction.
- Box duplication, absorption, croissant-box interaction. These operations each involve an edge of type $!A$, which is unaffected by the DS transformation. Thus, the identical operation can be performed in $\mathcal{D}(G)$.

To show that reductions in the bottom path can be simulated by reductions in the top path, we proceed by induction on the number of reduction steps from $\mathcal{D}(G)$ to G''. Since the DS transformation only converts λ- and apply nodes to edges, an edge e between two interaction ports in $\mathcal{D}(G)$ corresponds to a path p in G consisting of a sequence of lambda and apply nodes, such that $\mathcal{D}(p) = e$. If p is not an edge, we show that p must β-reduce to one; thus, a single reduction step in $\mathcal{D}(G)$ is simulated by a sequence of β-steps in G, followed by the same reduction step as performed in $\mathcal{D}(G)$. Because p consists of only λ- and apply nodes, it suffices to show that p is a balanced path. Consider each possible redex in $\mathcal{D}(G)$:

- A β-redex. If p were unbalanced, by Proposition 5, either the first or last edge would have type $A \to \bot$, for some A. In that case, the lambda or apply node connected to that edge would be eliminated by the DS transformation, contradicting the fact that p connects nodes that form a beta redex in $\mathcal{D}(G)$. Thus, p must be well-balanced.
- Box duplication, box absorption, croissant-box interaction. In all of these cases, the type of the edge between the interaction ports is $!A$. Because the boxes, croissants, and sharing nodes are not affected by the DS transformation, the first and last edges of p must also have type $!A$. By Proposition 4, p cannot be unbalanced. Thus, by Proposition 2, p must be well-balanced.

3.4 Embeddings of Classical Logic

Each of our proofnet implementations implicitly encodes, via an extended Curry-Howard correspondence, an embedding of classical logic in multiplicative-exponential linear logic (MELL). This family of encodings results from the mix-and-match of standard double-negation embeddings of classical logic into intuitionistic logic, composed with embeddings of intuitionistic logic into MELL. We discuss these constructions for minimal implicational logic with $\neg\neg$-elimination.

Let $[\alpha \to \beta] \equiv ![\alpha] \multimap [\beta]$ be the Girard translation of intuitionistic implication in linear logic; similarly, let $\langle \alpha \to \beta \rangle \equiv !(\langle \alpha \rangle \multimap \langle \beta \rangle)$ be the Gonthier-Abadi-Lévy translation (see [12]). By standard linear logic identities, $[\neg\neg\tau] = ?![\tau]$ and $\langle \neg\neg\tau \rangle = !?\langle \tau \rangle$, where ! and ? are the (dual) exponential modalities.

What does this mean in terms of graph reduction? When a subgraph G with root typed $?!\alpha$ is substituted into a sharable context C, the ? marks a croissant at the root of G that breaks the box around C, which then shares the value of type $!\alpha$. Dually, if G has type $!?\beta$, the context has type $?!(\beta)^{\perp}$, and the protocol for box opening and sharing reverses the role of context and value.

Recall $(\alpha \to \beta)^* \equiv \alpha^* \to \neg\neg\beta^*$ is the translation of $\alpha \to \beta$ induced by the CBV CPS transformation, and $(\alpha \to \beta)^\dagger \equiv \neg\neg\alpha^\dagger \to \neg\neg\beta^\dagger$ is the translation induced by the CBN CPS transformation. The CPS translations of a function (classical proof) $E : \alpha \to \beta$ result in a function of type $\neg\neg(\alpha \to \beta)^*$ or $\neg\neg(\alpha \to \beta)^\dagger$; we then have

$$\begin{array}{ll} [\neg\neg(\alpha \to \beta)^*] = ?!(!\,[\alpha^*] \multimap ?!\,[\beta^*]) & \langle\neg\neg(\alpha \to \beta)^*\rangle = !?!(\langle\alpha^*\rangle \multimap !?\langle\beta^*\rangle) \\ \left[\neg\neg(\alpha \to \beta)^\dagger\right] = ?!(!?!\left[\alpha^\dagger\right] \multimap ?!\left[\beta^\dagger\right]) & \langle\neg\neg(\alpha \to \beta)^\dagger\rangle = !?!(!?\langle\alpha^\dagger\rangle \multimap !?\langle\beta^\dagger\rangle) \end{array}$$

Other variants of this mix-and-match style are possible. Fewer modalities mean fewer boxes and greater implementation efficiency.

4 Related Work

We consider three areas of related work: other CPS transformations, other approaches to converting CPS programs back to direct style, and other connections between control operators and linear logic.

Optimizing the CPS transformation: The Plotkin CPS transformations create many "administrative" redexes involving the application of a continuation or continuation abstraction [24]. Our DS transformation converts administrative redexes into box-croissant redexes, adding a bureaucratic cost to optimal reduction [1,20]. More optimized CPS transformations [7,25] could generate more efficient implementations.

Converting CPS programs back to direct style: Danvy first investigated the problem of converting a CPS program back to direct style [5], later extended with Lawall. The conversions were only on terms that could be output by CPS transformation. Our DS transformation also relies on a uniform, but weaker property: values of type $\perp$ must occur consistently, and do not contribute to the final result. At the extreme, our DS transformation is simply the identity on the graphs of DS terms.

Relating languages with control operators to linear logic: Nishizaki also investigated encodings of λ-calculus plus `call/cc` in proofnets [22]. He showed that normalization of these proofnets is complete with respect to normalization in the term language. He began by adding modalities in an ad hoc manner (induced mechanically by our $\mathrm{CBV}_{\mathrm{CPS/N}}$ translation) to the type $!A \multimap B$, allowing sharing of both values and continuations. His more complex translation is an optimization of our $\mathrm{CBV}_{\mathrm{CPS/N}}$ translation, eliminating some box croissant interactions corresponding to administrative redexes. Because Nishizaki derived a translation from the types rather than from the semantics, he had to prove that the resulting graphs model the semantics of the language. The correctness of our approach relies only on the correctness of the CPS transformation, and on the correctness of the DS transformation on graphs, which is independent of the language being implemented.

In a sequel to his earlier work on symmetric λ-calculus, Filinski used linear logic as a tool for understanding continuations [9]. Some of the linear types he proposed for continuations appear in our codings, the most common being $?!\alpha$, resulting from the DS transformation of a graph with type $(\alpha \multimap \bot) \multimap \bot$. Griffin, and later Murthy, showed the relation between so-called $\neg\neg$-embeddings of classical logic in intuitionistic logic, and the implementation of control operators [14,21]. In particular, they showed how varieties of the CPS transformation provide the constructive content of such embeddings. We further translate such terms into proofnets in direct style, eliminating the administrative redexes. The result is a family of constructive embeddings of classical logic into linear logic.

5 Future Work

Since the continuation created by `call/cc` is abortive, a term containing `call/cc` can reduce to different normal forms; its CPS counterpart, like all pure λ-terms, has only one normal form. Because the DS transformation produces boxes according to the CPS transformation providing its input, it should be possible to identify, among the shared normal forms it can return, the answer that would have been produced by the CPS-converted input. We leave further analyses of these observations to future work.

Efficiently managing the reified continuation is a significant problem in implementing languages with control operators [4,16]. Proofnet implementations suggest the possibility of evaluating programs containing control operators using optimal reduction, with a minimal copying of shared values. Nevertheless, we are exchanging the savings of optimal reduction for the overhead of box management. Further experiments are needed to understand whether the exchange is cost-effective, and if it can be further optimized by better box technology.

Our proofnet technology might be extended to languages with functional control operators, such as Danvy and Filinski's `shift` and `reset`, and Sitaram and Felleisen's `control` and `prompt` [6,27]. While `shift` and `reset` are defined in terms of a CPS transformation, continuations do not have return type $\bot$; the DS transformation is then inapplicable. Both `shift` and `reset`, and `control` and `prompt` can be defined in terms of `call/cc` and a reference cell [10,27]. Bawden has shown how to implement reference cells using sharing graphs [3], so this strategy may still lead to an interesting proofnet implementation.

6 Conclusions

We have shown how to implement various languages with explicit control using graph reduction, where the structure of the graphs are proofnets from linear logic. The principal technical difficulty in such codings is the location of boxes, which allow computations to be shared. Rather than specifying a fixed scheme for locating boxes, we have introduced a general methodology based on the CPS transform. Different versions of CPS, followed by our DS transform on graphs, produce a wide range of consistent schemes for locating boxes in proofnets. The noble art of linear decorating, to repeat the phrase of Schellinx [26], has been replaced by a factory.

The theoretical foundation of our implementation technology means that we have a consistent semantics provided by the geometry of interaction, and a means of incrementally evaluating continuations via optimal evaluation. The codings may clarify full abstraction theorems for languages with explicit control, given the full completeness results that are known for linear logic. But the genuine progress reflected in the presented techniques is the technology transfer of logic and proofnets to the mundane algorithmics of implementation. The pragmatics of double negation in logic, for example, is just *packaging:* the boxing of sharable data so that they can interact with each other. Further implementation improvements amount to a better understanding of where to put boxes. A generation of compiler writers has spent considerable effort optimizing the efficiency of sharing expressions. We have presented a systematic basis on which to optimize the sharing of continuations, providing new territory for similar efficiency improvements.

Acknowledgments. We thank Alan Bawden and Olivier Danvy for commenting on a draft of this paper.

References

1. A. Asperti. $\delta \circ !\epsilon = 1$: Optimizing optimal λ-calculus implementations. In *Rewriting Techniques and Applications*, pages 102–116, Kaiserslautern, Germany, 1995.
2. A. Asperti and C. Laneve. Paths, computations and labels in the lambda-calculus. In *Rewriting Techniques and Applications, 5th International Conference*, volume 690, pages 152–167. Lecture Notes in Computer Science, 1993.
3. A. Bawden. Implementing distributed systems using linear naming. TR 1627, MIT AI Lab, March 1993. (PhD thesis, originally titled *Linear Graph Reduction: Confronting the Cost of Naming.* MIT, May 1992).
4. C. Bruggeman, O. Waddell, and R.K. Dybvig. Representing control in the presence of one-shot continuations. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation*, volume 31(5), pages 99–107, Philadephia, Pennsylvania, May 1996. SIGPLAN Notices.
5. O. Danvy. Back to direct style. *Science of Computer Programming*, 22(3):183–195, 1994.
6. O. Danvy and A. Filinski. Abstracting control. In *Proceeding of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, June 1990. ACM Press.
7. O. Danvy and A. Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 4:361–391, 1992.
8. A. Filinski. Declarative continuations and categorical duality. Technical Report 89/11, University of Copenhagen, 1989. Masters Thesis.
9. A. Filinski. Linear continuations. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 27–38, Albuquerque, New Mexico, January 1992.

10. A. Filinski. Representing monads. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 446–457, Portland, Oregon, January 1994.
11. J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 46:1–102, 1986.
12. G. Gonthier, M. Abadi, and J.-J. Lévy. The geometry of optimal lambda reduction. In *Conference record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 15–26, Albuquerque, New Mexico, January 1992.
13. G. Gonthier, M. Abadi, and J.-J. Lévy. Linear logic without boxes. In *Proceedings of the Seventh Annual Symposium on Logic in Computer Science*, pages 223–234, June 1992.
14. T. Griffin. A formulae-as-types notion of control. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 47–58, San Francisco, California, January 1990.
15. J. Hatcliff and O. Danvy. A generic account of continuation-passing styles. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 458–471, Portland, Oregon, January 1994.
16. R. Hieb, R.K. Dybvig, and C. Bruggeman. Representing control in the presence of first-class continuations. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, volume 25(6), pages 66–77, White Plains, New York, June 1990. SIGPLAN Notices.
17. J. Lamping. An algorithm for optimal lambda calculus reduction. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 16–30, San Francisco, California, January 1990.
18. J.-J. Lévy. *Optimal Reductions in the Lambda-Calculus*, pages 159–191. Academic Press, 1980.
19. I. Mackie. *The Geometry of Implementation.* PhD thesis, University of London, September 1994.
20. I. Mackie. YALE: Yet another lambda evaluator based on interaction nets. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 117–128, Baltimore, Maryland, September 1998.
21. C.R. Murthy. *Extracting Constructive Content from Classical Proofs.* PhD thesis, Cornell University, August 1990.
22. S. Nishizaki. Programs with continuations and linear logic. *Science of Computer Programming*, 21(2):165–190, 1993.
23. M. Parigot. Lambda-mu-calculus: an algorithmic interpretation of classical natural deduction. In A. Voronkov, editor, *Logic Programming and Automated Reasoning: International Conference LPAR '92 Proceedings, St. Petersburg, Russia*, pages 190–201, Berlin, DE, 1992. Springer-Verlag.
24. G. D. Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.
25. A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3/4):289–360, November 1993.
26. H. Schellinx. *The Noble Art of Linear Decorating.* PhD thesis, Institute for Logic, Language and Computation, University of Amsterdam, 1994.
27. D. Sitaram and M. Felleisen. Control delimiters and their hierarchies. *Lisp and Symbolic Computation*, 3(1):67–99, January 1990.

A Calculus for Link-Time Compilation

Elena Machkasova[1] and Franklyn A. Turbak[2]*

[1] Boston University, Boston MA 02215, USA,
elenam@bu.edu
[2] Wellesley College, Wellesley MA 02481, USA,
fturbak@wellesley.edu

Abstract. We present a module calculus for studying a simple model of link-time compilation. The calculus is stratified into a term calculus, a core module calculus, and a linking calculus. At each level, we show that the calculus enjoys a *computational soundness* property: if two terms are equivalent in the calculus, then they have the same outcome in a small-step operational semantics. This implies that any module transformation justified by the calculus is meaning preserving. This result is interesting because recursive module bindings thwart confluence at two levels of our calculus, and prohibit application of the traditional technique for showing computational soundness, which requires confluence. We introduce a new technique, based on properties we call *lift* and *project*, that uses a weaker notion of *confluence with respect to evaluation* to establish computational soundness for our module calculus. We also introduce the *weak distributivity property* for a transformation T operating on modules D_1 and D_2 linked by $\oplus$: $T(D_1 \oplus D_2) = T(T(D_1) \oplus T(D_2))$. We argue that this property finds promising candidates for link-time optimizations.

1 Introduction

We present a module calculus for a purely functional language that is a tool for exploring the design space for a simple form of link-time compilation. Link-time compilation lies in the relatively unexplored expanse between *whole-program compilation*, in which the entire source program is compiled to an executable, and *separate compilation*, in which source program modules are independently compiled into fragments, which are later linked to form an executable. In the link-time compilation model (1) source program modules are first partially compiled into intermediate language modules; (2) intermediate modules are further compiled when they are combined, taking advantage of usage information exposed by the combination; and (3) when all intermediate modules have been combined into a final closed module, it is translated into an executable.

Link-time compilation can potentially provide more reusability than whole-program compilation and more efficiency than separate compilation. While separate compilation offers well-known benefits for program development and code

* Both authors were supported by NSF grant EIA–9806747. This work was conducted as part of the Church Project (http://www.cs.bu.edu/groups/church/).

reuse, a drawback is that the compilation of one module cannot take advantage of usage information in the modules with which it is later linked. In contrast, link-time compilation can use this information to perform optimizations and choose specialized data representations more efficient than the usual uniform representations for data passed across module boundaries.

In this paper we take some first steps towards formalizing link-time compilation. There are three main contributions of this work. First, we present a stratified untyped call-by-value module calculus that at every level satisfies a *computational soundness* property[1]: if two expressions can be shown equivalent via calculus steps, then their outcomes relative to a small-step operational semantics will be observably equal. This implies that any transformation expressible as a sequence of calculus steps (such as constant propagation and folding, function inlining, and many others) is meaning preserving.

Second, our technique for proving soundness is interesting in its own right. Traditional techniques for showing this property (e.g., [Plo75,AF97]) require the language to be confluent, but the recursive nature of module bindings destroys confluence. In order to show that our module calculus has soundness, we introduce a new technique for proving this property based on a weaker notion we call *confluence with respect to evaluation*. We replace the confluence and standardization of the traditional technique for proving soundness with symmetric properties we call *lift* and *project*.

Third, we sketch a simple model of link-time compilation and introduce the *weak distributivity* property as one way to find candidates for link-time optimizations. We show that module transformations satisfying certain conditions are weakly distributive, and demonstrate these conditions for some examples of meaning preserving transformations.

Our work follows a long tradition of using untyped calculi for reasoning about programming languages features: e.g., call-by-name vs. call-by-value semantics [Plo75], call-by-need semantics [AFM$^+$95,AF97], state and control [FH92], and sharing and cycles [AK97,AB97]. Our notion of confluence with respect to evaluation avoids cyclic substitutions in the operational semantics, and so is related to the acyclic substitution restriction of Ariola and Klop [AK97].

This work is part of a renewed interest in linking issues that was inspired by Cardelli's call to arms [Car97]. Recent work on module systems and linking has focused on such issues as: sophisticated type systems for modules [HL94,Ler94]; the expressiveness of modules systems (e.g., handling features like recursive modules [FF98,CHP97,AZ99], inheritance and mixins [DS96,AZ99] and dynamic linking [FF98,WV99]); binary compatibility in the context of program modifications [SA93,DEW99]; and modularizing module systems [Ler96,AZ99]. There has been relatively little focus on issues related to link-time optimization; exceptions are [Fer95] and recent work on just-in-time compilers (e.g, [PC97]).

Our work stands out from other work on modules in two important respects. First, we partition the reduction relation of the calculus ($\rightarrow$) into *evaluation* (sometimes called *standard*) steps ($\Rightarrow$) that define a small-step operational se-

[1] We will often abbreviate the name of this property as "soundness".

mantics and *non-evaluation* (*non-standard*) steps ($\hookrightarrow$). While this partitioning is common in the calculus world (e.g., [Plo75,FH92,AF97]), it is rare in the module world. Typical work on modules (e.g., [Car97,AZ99]) gives only an operational semantics for modules. Yet in the context of link-time compilation, the notion of reduction in a calculus is essential for justifying meaning preserving program transformations. Without non-evaluation steps, even simple transformations like transforming $[F \mapsto \lambda x.(1+2)]$ to $[F \mapsto \lambda x.3]$ or $[A \mapsto 4, F \mapsto \lambda x.x + A]$ to $[A \mapsto 4, F \mapsto \lambda x.x + 4]$ are difficult to prove meaning preserving.

Second, unlike most recent work on modules (with the notable exception of [WV99]), our work considers only an untyped module language. There are several reasons for this. First, types are orthogonal to our focus on computational soundness and weak distributivity; types would only complicate the presentation. Second, introducing types often requires imposing restrictions that we would like to avoid. For example, to add types to their system, [AZ99] need to impose several restrictions on their untyped language: no components with recursive types, and no modules as components to other modules. Finally, we do not yet have anything new to say in the type dimension. We believe that it is straightforward to adapt an existing simple module type system (e.g., [Car97,FF98,AZ99]) to our calculus. On the other hand, we think that enriching our module system with polymorphic types is a very interesting avenue for future exploration.

Due to space limitations, our presentation is necessarily dense and telegraphic. Please see the companion technical report [MT00] for a more detailed exposition with additional explanatory text, more examples, and proofs.

2 The Module Calculus

In this section, we present a stratified calculus with three levels: a term calculus $\mathcal{T}$, a core module calculus $\mathcal{C}$, and a full module calculus $\mathcal{F}$. The three calculi are summarized in Fig. 1. Let $\mathcal{X}$ range over $\{\mathcal{T}, \mathcal{C}, \mathcal{F}\}$. The definition for each calculus $\mathcal{X}$ consists of the following:

- The syntax for calculus terms $\mathbf{Term}_{\mathcal{X}}$ and for general one-hole contexts $\mathbf{Context}_{\mathcal{X}}$. If $\mathbb{X} \in \mathbf{Context}_{\mathcal{X}}$, then $\mathbb{X}\{Y\}$ denotes the result of filling the hole of $\mathbb{X}$ with a term Y. Due to the hierarchical structure of our module calculus, Y is not necessarily a term of $\mathcal{X}$. For instance, in our hierarchy $\mathcal{T}$ contexts are filled with $\mathcal{T}$ terms; $\mathcal{C}$ contexts are filled with $\mathcal{T}$ terms; and $\mathcal{F}$ contexts are filled with either $\mathcal{C}$ or $\mathcal{F}$ terms. We assume that the notation $\mathbb{X}\{Y\}$ is only applied to such $\mathbb{X}$ and Y that the result of the filling is a well-formed term in $\mathbf{Term}_{\mathcal{X}}$. For instance, the notation $\mathbb{D}\{M\}$ is defined only if the resulting module is well-defined element of $\mathbf{Term}_{\mathcal{C}}$.
- A small-step operational semantics of $\mathcal{X}$ defined via an evaluation step relation $\Rightarrow_{\mathcal{X}}$, and a complementary definition of a non-evaluation step relation $\hookrightarrow_{\mathcal{X}}$. For each of the three calculi we define a one-step calculus relation $\rightarrow_{\mathcal{X}} \overset{\text{def}}{=} \Rightarrow_{\mathcal{X}} \cup \hookrightarrow_{\mathcal{X}}$.[2] The relation $\Rightarrow_{\mathcal{X}}$ is often defined in terms of an *evalua-*

[2] Alternatively we could have defined the rules for $\rightarrow_{\mathcal{X}}$ explicitly and then set $\hookrightarrow_{\mathcal{X}}$ to be $\rightarrow_{\mathcal{X}} \setminus \Rightarrow_{\mathcal{X}}$. However, giving explicit rules for $\hookrightarrow_{\mathcal{X}}$ clarifies the presentation.

Syntax for the Term Calculus ($\mathcal{T}$):

$$\begin{array}{ll} c \in \mathbf{Const} \;=\; \textit{constant values} & x \in \mathbf{Variable} \;=\; \textit{term variables} \\ v \in \mathbf{Visible} \;=\; \textit{external labels} & h \in \mathbf{Hidden} \;=\; \textit{internal labels} \end{array}$$

$$k, l \in \mathbf{Label} \;=\; \mathbf{Visible} \cup \mathbf{Hidden}$$

$$L, M, N \in \mathbf{Term}_{\mathcal{T}} \;::=\; c \mid x \mid l \mid (\lambda x.M) \mid M_1 \mathbin{@} M_2 \mid M_1 \mathbin{\widehat{\mathrm{op}}} M_2$$

$$\mathbb{C} \in \mathbf{Context}_{\mathcal{T}} \;::=\; \Box \mid (\lambda x.\mathbb{C}) \mid \mathbb{C} \mathbin{@} M \mid M \mathbin{@} \mathbb{C} \mid \mathbb{C} \mathbin{\widehat{\mathrm{op}}} M \mid M \mathbin{\widehat{\mathrm{op}}} \mathbb{C}$$

$$V \in \mathbf{Value}_{\mathcal{T}} \;::=\; c \mid x \mid \lambda x.M$$

Notion of Reduction on Terms:

$$\begin{array}{rll} (\lambda x.M \mathbin{@} V) & \leadsto_{\mathcal{T}} M[x := V] & (\beta) \\ c_1 \mathbin{\widehat{\mathrm{op}}} c_2 & \leadsto_{\mathcal{T}} c, \text{where } c = \delta(\widehat{\mathrm{op}}, c_1, c_2) & (\delta) \end{array}$$

Evaluation and Non-evaluation Steps:

$$\mathbb{E} \in \mathbf{EvalContext}_{\mathcal{T}} \;::=\; \Box \mid \mathbb{E} \mathbin{@} M \mid (\lambda x.M) \mathbin{@} \mathbb{E} \mid \mathbb{E} \mathbin{\widehat{\mathrm{op}}} M \mid c \mathbin{\widehat{\mathrm{op}}} \mathbb{E}$$

$$\begin{array}{llll} \mathbb{E}\{R\} \Rightarrow_{\mathcal{T}} \mathbb{E}\{Q\}, & \text{where} & R \leadsto_{\mathcal{T}} Q, & \text{(term-ev)} \\ \overline{\mathbb{E}}\{R\} \hookrightarrow_{\mathcal{T}} \overline{\mathbb{E}}\{Q\}, & \text{where} & R \leadsto_{\mathcal{T}} Q. & \text{(term-nev)} \end{array}$$

Syntax for the Core Module Calculus ($\mathcal{C}$):

$$D \in \mathbf{Term}_{\mathcal{C}} \;::=\; [l_1 \mapsto M_1, \ldots, l_n \mapsto M_n] \quad (\text{abbreviated } [l_i \overset{n}{\underset{i=1}{\mapsto}} M_i]),$$

provided $l_i = l_j$ implies $i = j$, $FV(D) = \emptyset$, and $\mathit{Imports}(D) \cap \mathbf{Hidden} = \emptyset$.

$$\mathbb{D} \in \mathbf{Context}_{\mathcal{C}} \;::=\; [l_i \overset{k-1}{\underset{i=1}{\mapsto}} M_i, l_k \mapsto \mathbb{C}, l_j \overset{n}{\underset{j=k+1}{\mapsto}} M_j]$$

Projection Notation: $[l_i \overset{n}{\underset{i=1}{\mapsto}} M_i] \downarrow l_j = M_j$, if $1 \leq j \leq n$, and otherwise undefined.

Evaluation and Non-evaluation Steps:

$$\mathbb{G} \in \mathbf{EvalContext}_{\mathcal{C}} ::= [l_i \overset{k-1}{\underset{i=1}{\mapsto}} M_i, l_k = \mathbb{E}, l_j \overset{n}{\underset{j=k+1}{\mapsto}} M_j]$$

$$\begin{array}{rll} \mathbb{G}\{R\} \Rightarrow_{\mathcal{C}} & \mathbb{G}\{Q\}, \text{ where } R \leadsto_{\mathcal{T}} Q. & \text{(comp-ev)} \\ \mathbb{G}\{l\} \Rightarrow_{\mathcal{C}} & \mathbb{G}\{V\}, \text{ where } \mathbb{G}\{l\} \downarrow l = V. & \text{(subst-ev)} \\ [l_i \overset{n}{\underset{i=1}{\mapsto}} M_i, h_j \overset{m}{\underset{j=1}{\mapsto}} V_j] \Rightarrow_{\mathcal{C}} & [l_i \overset{n}{\underset{i=1}{\mapsto}} M_i], \text{ where } \forall_{1 \leq i \leq m}.h_i \notin \cup_{j=1}^{n} FL(M_j) & \text{(GC)} \\ \overline{\mathbb{G}}\{R\} \hookrightarrow_{\mathcal{C}} & \overline{\mathbb{G}}\{Q\}, \text{ where } R \leadsto_{\mathcal{T}} Q. & \text{(comp-nev)} \\ \overline{\mathbb{G}}\{l\} \hookrightarrow_{\mathcal{C}} & \overline{\mathbb{G}}\{V\}, \text{ where } \overline{\mathbb{G}}\{l\} \downarrow l = V. & \text{(subst-nev)} \end{array}$$

Syntax for the Full Module Calculus ($\mathcal{F}$):

$$F \in \mathbf{Term}_{\mathcal{F}} ::= D \mid I \mid F_1 \oplus F_2 \mid F[l \leftarrow l'] \mid \mathbf{let}\ I = F_1\ \mathbf{in}\ F_2$$

$$\mathbb{F} \in \mathbf{Context}_{\mathcal{F}} ::= \Box \mid \mathbb{F} \oplus F \mid F \oplus \mathbb{F} \mid \mathbb{F}[l \leftarrow l'] \mid \mathbf{let}\ I = \mathbb{F}\ \mathbf{in}\ F \mid \mathbf{let}\ I = F\ \mathbf{in}\ \mathbb{F}$$

Evaluation and Non-evaluation Steps:

$$\begin{array}{rll} D \Rightarrow_{\mathcal{F}} & D', \text{ where } D \Rightarrow_{\mathcal{C}} D' & \text{(mod-ev)} \\ \mathbb{F}\{[k_i \overset{n}{\underset{i=1}{\mapsto}} M_i] \oplus [l_j \overset{m}{\underset{j=1}{\mapsto}} N_j]\} \Rightarrow_{\mathcal{F}} & \mathbb{F}\{[k_i \overset{n}{\underset{i=1}{\mapsto}} M_i, l_j \overset{m}{\underset{j=1}{\mapsto}} N_j]\}, & \text{(link)} \\ & \text{where } (\cup_{i=1}^{n} k_i) \cap (\cup_{j=1}^{m} l_j) = \emptyset & \\ \mathbb{F}\{D[l \leftarrow k]\} \Rightarrow_{\mathcal{F}} & \mathbb{F}\{D[l := k]\}, & \text{(rename)} \\ & \text{where } l \in BL(D) \text{ implies } k \notin BL(D), & \\ & l \in \mathbf{Hidden} \text{ implies } k \in \mathbf{Hidden}, \text{ and} & \\ & k \in \mathbf{Hidden} \text{ implies } l \notin \mathit{Imports}(D). & \\ \mathbb{F}\{\mathbf{let}\ I = F_1\ \mathbf{in}\ F_2\} \Rightarrow_{\mathcal{F}} & \mathbb{F}\{F_1[I := F_2]\}, & \text{(let)} \\ \mathbb{F}\{D\} \hookrightarrow_{\mathcal{F}} & \mathbb{F}\{D'\}, \text{ where } D \rightarrow_{\mathcal{C}} D' & \text{(mod-nev)} \\ & \text{and } \mathbb{F} \neq \Box \text{ or } D \hookrightarrow_{\mathcal{C}} D' & \end{array}$$

Fig. 1. The three levels of the module calculus.

> *tion context* $\mathbf{EvalContext}_{\mathcal{X}} \subseteq \mathbf{Context}_{\mathcal{X}}$. A term Y is a $\rightarrow_{\mathcal{X}}$*-normal-form (NF)* if there is no term N s.t. $M \rightarrow_{\mathcal{X}} N$, a $\Rightarrow_{\mathcal{X}}$*-NF* is defined analogously. For each calculus $\mathcal{X}$, there is a classification function $Cl_{\mathcal{X}}$ that maps each term to a "class" token that describes its state w.r.t. evaluation. The classes for evaluatable terms must be disjoint from those in $\Rightarrow_{\mathcal{X}}$-NF. Also associated with each calculus $\mathcal{X}$ is a set $\mathbf{Value}_{\mathcal{X}}$ of *values* that is the union of one or more classes of $\Rightarrow_{\mathcal{X}}$-NFs. The function $Outcome_{\mathcal{X}}$ of a term is defined to be the class of its $\Rightarrow$-normal form or a symbol $\perp$ if the term diverges.

We use the following notations and conventions. If $\mathbb{X}$ ranges over $\mathbf{EvalContext}_{\mathcal{X}}$, then $\overline{\mathbb{X}}$ ranges over $\mathbf{Context}_{\mathcal{X}} \setminus \mathbf{EvalContext}_{\mathcal{X}}$ (i.e. the set of *non-evaluation contexts*). For pairs of rules such as (comp-ev) and (comp-nev), which only differ by the use of an evaluation versus a non-evaluation context, we introduce a notation for the combined calculus rule. For instance, we say that $D \rightarrow_{\mathcal{C}} D'$ by the rule (comp) if either $D \Rightarrow_{\mathcal{C}} D'$ by (comp-ev) or $D \hookrightarrow_{\mathcal{C}} D'$ by (comp-nev). If $\rightarrow$ is a one-step relation, then $\rightarrow^*$ denotes its reflexive transitive closure, and $\leftrightarrow$ denotes its reflexive, symmetric, and transitive closure.

The following properties of calculi are important in the sequel.

Definition 1 (Confluence). *The* $\rightarrow$ *relation is confluent if* $M_1 \rightarrow^* M_2$ *and* $M_1 \rightarrow^* M_3$ *implies the existence of* M_4 *s.t.* $M_2 \rightarrow^* M_4$, $M_3 \rightarrow^* M_4$. *A calculus* X *has confluence if* $\rightarrow_{\mathcal{X}}$ *is confluent.*

Definition 2 (Standardization). *A calculus* $\mathcal{X}$ *has the standardization property if for any sequence* $M_1 \rightarrow_{\mathcal{X}}^* M_2$ *there exists* M_3 *s.t.* $M_1 \Rightarrow_{\mathcal{X}}^* M_3 \hookrightarrow^*{}_{\mathcal{X}} M_2$.

2.1 Term Calculus ($\mathcal{T}$)

The module calculus is built on top of a term calculus $\mathcal{T}$, a typical call-by-value λ-calculus that includes constants (which we assume include integers) and binary operators (we assume $\hat{\mathrm{op}}$ includes standard integer operations). For interfacing with the module language in which it is embedded, the term syntax also includes two disjoint classes of labels whose union, **Label**, is itself disjoint from **Variable**.

We adopt the convention that all λ-bound variables in a term must be distinct. The free variables of a term M, written $FV(M)$, are defined as usual (recall that variables are distinct from labels). The set of labels appearing in a term M is written $FL(M)$; because labels cannot be λ-bound, they always appear "free". The result of a capture-avoiding substitution of M' for x in M is written $M[x := M']$. In addition to using α-renaming to avoid variable capture during substitution, it may be necessary to α-rename the result of substitution to maintain the distinct variable naming invariant. The result of substituting a term M' for a label l in M is written $M[l := M']$.

Both $\Rightarrow_{\mathcal{T}}$ and $\hookrightarrow_{\mathcal{T}}$ are defined via a redex/contractum relation $\leadsto_{\mathcal{T}}$ specified by a call-by-value β rule and a δ rule (unspecified) for binary functions on constants. Terms in $dom(\leadsto_{\mathcal{T}})$ are called *term redexes*. The relations $\Rightarrow_{\mathcal{T}}$ and $\hookrightarrow_{\mathcal{T}}$ are contextual closures of $\leadsto_{\mathcal{T}}$ with respect to an *evaluation context* $\mathbb{E}$ and a *non-evaluation context* $\overline{\mathbb{E}}$. It is easy to see that $\rightarrow_{\mathcal{T}}$ (defined as $\Rightarrow_{\mathcal{T}} \cup \hookrightarrow_{\mathcal{T}}$) is the contextual closure of $\leadsto_{\mathcal{T}}$ with respect to a general context $\mathbb{C}$.

A term M can be uniquely classified with respect to evaluation via $Cl_{\mathcal{T}}(M)$, defined as:

$$\begin{array}{llllll} \mathbf{const}(c) & \text{if } M = c & \mathbf{abs} & \text{if } M = \lambda x.N & \mathbf{evaluatable} & \text{if } M = \mathbb{E}\{R\} \\ \mathbf{var} & \text{if } M = x & \mathbf{stuck}(l) & \text{if } M = \mathbb{E}\{l\} & \mathbf{error} & \text{otherwise} \end{array}$$

It turns out that an evaluatable term M can be uniquely parsed into $\mathbb{E}$ and R such that $M = \mathbb{E}\{R\}$, so $\Rightarrow_{\mathcal{T}}$ is deterministic (i.e., it is a partial function rather than a relation). The partial function $Eval_{\mathcal{T}}(M)$ is defined as the $\Rightarrow_{\mathcal{T}}$-NF of M if it exists; otherwise, M is said to diverge. The total function $Outcome_{\mathcal{T}}(M)$ is defined as $Cl_{\mathcal{T}}(Eval_{\mathcal{T}}(M))$ if $Eval_{\mathcal{T}}(M)$ is defined, and $\perp$ if M diverges. Using classical techniques [Plo75,Bar84], it is straightforward to prove that $\rightarrow_{\mathcal{T}}$ is confluent, and $\mathcal{T}$ has the standardization property.

2.2 Core Module Calculus ($\mathcal{C}$)

In our module calculus, modules are unordered collections of labeled terms. There are two disjoint classes of labels: *visible* and *hidden*. Visible labels name components to be exported to other modules, and also name import sites within a component, while hidden labels name components that can only be referenced within the module itself. (This distinction is similar to distinction between deferred variables and expression names on one hand and local variables on the other in [AZ99]). Intuitively, a module is a fragment of a recursively scoped record that can be dynamically constructed by linking, where visible labels serve to "wire" the definitions in one module to the uses in another.

A *module binding* is written $l \mapsto M$. A *module* is a bracketed set of such bindings in which the labels of any two bindings are distinct. Note that a hole in a module context $\mathbb{D}$ is filled with a $\mathcal{T}$-term rather than another module. The notation $l_i \overset{n}{\underset{i=1}{\mapsto}} M_i$ stands for the bindings $l_1 \mapsto M_1 \ldots l_n \mapsto M_n$, and $D \downarrow l$ extracts the component M bound to l in D (if it exists).

Suppose that $D = [l_i \overset{n}{\underset{i=1}{\mapsto}} M_i]$. The free variables of D are $FV(D) = \cup_{i=1}^{n} FV(M_i)$. The substitution $D[l := k]$ yields $[l'_i \overset{n}{\underset{i=1}{\mapsto}} M_i[l := k]]$, where $l'_i = k$ if $l_i = l$ and $l'_i = l_i$ otherwise. The set of *bound labels* in D is defined as $BL(D) = \cup_{i=1}^{n} l_i$, while the set of *free labels* is $FL(D) = (\cup_{i=1}^{n} FL(M_i))/BL(D)$. The *exported labels* of D are those that are both bound and visible ($Exports(D) = BL(D) \cap \mathbf{Visible}$), while the *imported labels* are just the free ones ($Imports(D) = FL(D)$). In order to be well-formed, a module D must satisfy three conditions: (1) all its bound labels must be distinct; (2) it must not import any hidden labels; and (3) it must not contain any free variables (such variables would necessarily be unbound). In a well-formed module, the hidden labels are necessarily bound, so we define $Hid(D) = BL(D) \cap \mathbf{Hidden}$.

The evaluation relation $\Rightarrow_{\mathcal{C}}$ is defined using a *module evaluation context* $\mathbb{G}$ which lifts term-level evaluation context $\mathbb{E}$ to the module level. The three rules of $\Rightarrow_{\mathcal{C}}$ allow the following reductions: (comp-ev) lifts $\Rightarrow_{\mathcal{T}}$ to the module level; (subst-ev) substitutes a labeled value for a label occurrence in the module; (GC) garbage collects hidden values not referenced elsewhere in the module. Unlike $\Rightarrow_{\mathcal{T}}$, $\Rightarrow_{\mathcal{C}}$ is not deterministic, because it can perform an evaluation step on any component. Nevertheless, $\Rightarrow_{\mathcal{C}}$ is confluent. The complementary relation $\hookrightarrow_{\mathcal{C}}$ has two rules (comp-nev) and (subst-nev) which differ from their evaluation analogs by using a non-evaluation context in place of an evaluation context. Note that the (GC) rule does not have a non-evaluation counterpart; i.e., all (GC)-reductions are evaluation steps.

Let us consider some examples of module reductions.[3] Any one-step reduction on a term component can be lifted to the module via the (comp) rule: $[F \mapsto \lambda x.1+2] \hookrightarrow_{\mathcal{C}} [F \mapsto =\lambda x.3]$. This is a non-evaluation step, since the redex occurs under a λ. As an example of (subst), consider $[A \mapsto 4, F \mapsto A+3] \Rightarrow_{\mathcal{C}} [A \mapsto 4, F \mapsto 4+3]$. Here A in the second term appears in an evaluation context. Note that a value may be substituted into itself: $[F \mapsto \lambda x.F] \hookrightarrow_{\mathcal{C}} [F \mapsto \lambda x.(\lambda x_1.F)] \hookrightarrow_{\mathcal{C}} [F \mapsto \lambda x.(\lambda x_1.(\lambda x_2.(\lambda x_3.F)))]$ (where α-renaming preserves the distinct variable invariant). This is a non-evaluation step, since F appears under a λ. The (GC) rule garbage collects hidden values not referenced elsewhere in the module. Consider:

$$\begin{aligned}&[P \mapsto \lambda w.g\ @\ (w+1), f \mapsto \lambda x.h, g \mapsto \lambda y.y*2, h \mapsto \lambda z.f]\\ &\quad \Rightarrow_{\mathcal{C}} [P \mapsto \lambda w.g\ @\ (w+1), g \mapsto \lambda y.y*2]\end{aligned}$$

The mutually recursive bindings for f and h can be removed because all references to these hidden labels occur inside of the values named by these labels. However, g cannot be removed, since an exported term references it.

It turns out that $\mathcal{C}$ has the standardization property. But interestingly, even though $\Rightarrow_{\mathcal{C}}$ is confluent, $\rightarrow_{\mathcal{C}}$ is *not* confluent, due to the possibility of mutually recursive (subst) redexes that appear under a λ and therefore not in an evaluation context. Consider an example due to [AK97]: $D_0 = [F \mapsto \lambda x.G, G \mapsto \lambda y.F]$. Then $D_0 \hookrightarrow_{\mathcal{C}} [F \mapsto \lambda x.\lambda y'.F, G \mapsto \lambda y.F] = D_1$ and $D_0 \hookrightarrow_{\mathcal{C}} [F \mapsto \lambda x.G, G \mapsto \lambda y.\lambda x'.G] = D_2$. D_1 (resp. D_2) has an even (resp. odd) number of λs for F and an odd (resp. even) number for G, and in every reduction sequence starting with D_1 (resp. D_2), all terms will have this property. Clearly, reduction sequences starting at D_1 and D_2 can never meet at a common term.

The confluence of $\Rightarrow_{\mathcal{C}}$ gives rise to a partial function $Eval_{\mathcal{C}}(D)$ that, when defined, returns a module whose components are all $\Rightarrow_{\mathcal{T}}$-normal forms. The classification notion also lifts to the module level: $Cl_{\mathcal{C}}(D) = [l_i \overset{n}{\underset{i=1}{\mapsto}} Cl_{\mathcal{T}}(M_i)]$, where $D = [l_i \overset{n}{\underset{i=1}{\mapsto}} M_i]$. As in the term calculus, $Outcome_{\mathcal{C}}(D) = Cl_{\mathcal{C}}(Eval_{\mathcal{C}}(D))$ if $Eval_{\mathcal{C}}(D)$ exists, and $\bot$ otherwise. We say that $D = [l_i \overset{n}{\underset{i=1}{\mapsto}} V_i]$ is a *module value* ($D \in$ $\mathbf{Value}_{\mathcal{C}}$) if $Hid(D) = \emptyset$ and $V_i \in \mathbf{Value}_{\mathcal{T}}$ for all $1 \leq i \leq n$.

[3] In examples, we adopt the convention that visible labels have uppercase names while hidden labels have lowercase names.

2.3 Full Module Calculus ($\mathcal{F}$)

The full module calculus extends the core module calculus with three module operators: linking, renaming, and binding. Intuitively, the linking of modules D_1 and D_2, written $D_1 \oplus D_2$, takes the union of their bindings. To avoid naming conflicts between both visible and hidden labels, $BL(D_1)$ and $BL(D_2)$ must be disjoint. The fact that the import labels of a well-formed module may not be hidden prevents the components of one module from accessing hidden components of another when they are linked.

The renaming operator renames any module label (visible or hidden, import or export). Renaming import and export labels is the way to connect an exported component of one module to an import site in another. Renaming a visible label to a fresh hidden label hides a component; a user-level "hiding" operator could be provided as syntactic sugar for such renaming. Finally, renaming of hidden variables to other hidden variables is necessary to guarantee that hidden variables are disjoint when they are linked. The side conditions on renaming prevents certain undesirable scenarios: (1) attempting to rename one bound label to another (causing a name clash); (2) renaming a hidden variable to a visible one, thereby exposing it; and (3) renaming a (necessarily visible) import to a hidden label, thereby making the module ill-formed.

The binding operator **let** $I = F_1$ **in** F_2 names the (result of evaluating the) definition term F_1 and uses the name within the body term F_2. This models situations in which the same module is used multiple times in different contexts.

The disjoint hidden label requirement for $\oplus$ simplifies reasoning about the calculus, but is severe from the perspective of a user, who should not be able to predict the names of the hidden labels of any module. We address this problem by supplying a user-level linking operator $\overline{\oplus}$ that can be defined in terms of the primitive linking operator $\oplus$ and renaming, as follows. Suppose that $F_1 = [v_i \overset{n_1}{\underset{i=1}{\mapsto}} M_i, h_j \overset{m_1}{\underset{j=1}{\mapsto}} N_j]$ and $F_2 = [v'_i \overset{m_2}{\underset{i=1}{\mapsto}} M'_i, h'_j \overset{n_2}{\underset{j=1}{\mapsto}} N'_j]$. Then $F_1 \overline{\oplus} F_2$ is defined as:

$$F_1[h_1 \leftarrow h''_1, \ldots, h_{n_1} \leftarrow {h_{n_1}}''] \;\oplus\; F_2[h'_1 \leftarrow {h_{n_1+1}}'', \ldots, h'_{n_2} \leftarrow {h_{n_1+n_2}}''],$$
$$\text{where } \left((\textstyle\bigcup_{i=1}^{n_1} h_i) \cup (\bigcup_{j=1}^{n_2} {h_j}')\right) \bigcap \left(\textstyle\bigcup_{k=1}^{n_1+n_2} {h_k}''\right) = \emptyset$$

The hidden labels of F_1 and F_2 are renamed to fresh hidden labels before the modules are linked to avoid collisions. The renaming performed by $\overline{\oplus}$ is is similar to the α-renaming required in other module calculi linking operations (e.g., in [FF98] when rewriting the **compound** linking form to the **unit** module form).

The definition of $\rightarrow_{\mathcal{F}}$ lifts core module reduction steps to the module expression level and adds evaluation rules for the link-level operators (link, rename, and bind). The structure of $\mathbf{Context}_{\mathcal{F}}$ allows the link-level operators to be evaluated in any order. The lifted core module reduction steps are only considered evaluation steps if they are not surrounded by any link-level operators; this forces all link-level steps to be performed first in a "link-time stage", followed by a "run-time stage" of core module steps.

The lack of confluence of $\rightarrow_{\mathcal{C}}$ is inherited by $\rightarrow_{\mathcal{F}}$, but we are still able to show that $\Rightarrow_{\mathcal{F}}$ is confluent and $\mathcal{F}$ has the standardization property. If F is a link, rename, or bind term, we define $Cl_{\mathcal{F}}(F)$ to be **linkable**; otherwise we define $Cl_{\mathcal{F}}(F)$ to be $Cl_{\mathcal{C}}(F)$ (in this case, $F \in \mathbf{Term}_{\mathcal{C}}$). $Outcome_{\mathcal{F}}$ is defined analogously with $Outcome_{\mathcal{C}}$, and $\mathbf{Value}_{\mathcal{F}} = \mathbf{Value}_{\mathcal{C}}$.

3 Meaning Preservation

The calculus defined in the previous section allows us to reason about module transformations. A *transformation* T of a calculus $\mathcal{X}$ is a relation $T : \mathcal{X} \times \mathcal{X}$. Even though T in general is not a function, we sometimes write $Z = T(Y)$ if $(Y, Z) \in T$. Below we define a notion of observational equivalence and, based on it, a notion of a meaning preserving transformation.

Definition 3 (Observational Equivalence). *Two terms Y and Z of a calculus $\mathcal{X}'$ are observationally equivalent in a calculus $\mathcal{X}$ (written $Y \cong_{\mathcal{X}} Z$) if for all contexts $\mathbb{X}$ s.t. $\mathbb{X}\{Y\}$ and $\mathbb{X}\{Z\}$ are well-formed terms of $\mathcal{X}$, $\mathbb{X}\{Y\} \Rightarrow^*_{\mathcal{X}} W$ iff $\mathbb{X}\{Z\} \Rightarrow^*_{\mathcal{X}} W'$ where W and $W' \in \mathbf{Value}_{\mathcal{X}}$ and $Cl_{\mathcal{X}}(W) = Cl_{\mathcal{X}}(W')$.*

In the definition, note that $\mathcal{X}$ may or may not be the same as $\mathcal{X}'$. As an example, two core modules are observationally equivalent in $\mathcal{F}$ if in any full module context $\mathbb{F}$ they evaluate to module values of the same class, as defined above. For instance, consider the following modules: $D_1 = [F \mapsto \lambda x.x + a, a \mapsto 1 + 2]$, $D_1' = [F \mapsto \lambda x.x + 3, a \mapsto 3]$, $D_2 = [S \mapsto N_1 + N_2]$, and $D_2' = [S \mapsto N_2 + N_1]$. $D_1 \cong_{\mathcal{F}} D_1'$ because the exported F behaves like an "add 3" function for both modules in any context. Assuming that $+$ is commutative, $D_2 \cong_{\mathcal{F}} D_2'$ because they evaluate to the same module value when they are placed in a context that supplies integer values for N_1 and N_2, and none of the two modules evaluates to a module value if the context does not supply such values.

Definition 4 (Meaning Preservation). *A transformation T of a calculus X' is meaning preserving in a calculus $\mathcal{X}$ if $(Y, Z) \in T$ implies $Y \cong_{\mathcal{X}} Z$.*

For instance, the constant folding/propagation transformation CFP in $\mathcal{C}$ is meaning preserving in $\mathcal{F}$, as seen in the above example with D_1 and D_1'. The example with D_2 and D_2' illustrates that a transformation SPO that swaps the operands of $+$ in $\mathcal{C}$ is also meaning preserving in $\mathcal{F}$.

3.1 Computational Soundness

Proving that a transformation is meaning preserving can be difficult and tedious work. However, if T is a *calculus-based transformation* in $\mathcal{X}$, i.e. $Y \leftrightarrow_{\mathcal{X}} Z$ for all $(Y, Z) \in T$, then it is automatically meaning preserving in a calculus $\mathcal{X}'$ satisfying the conditions of Lemma 1 below.

A key notion for showing the meaning preservation of calculus-based transformations is computational soundness:

Definition 5 (Computational Soundness). *A calculus $\mathcal{X}$ is computationally sound if $M \leftrightarrow_{\mathcal{X}} N$ implies $Outcome_{\mathcal{X}}(M) = Outcome_{\mathcal{X}}(N)$, where $M, N \in$* $\mathbf{Term}_{\mathcal{X}}$.

It follows from computational soundness that if two terms are equivalent in the calculus then they are observationally equivalent in an *empty* context. For observationally equivalence to hold in *all* contexts requires *embedding*:

Definition 6 (Embedding). *A relation $\rightarrow_{\mathcal{X}'}$ is embedded in a relation $\rightarrow_{\mathcal{X}}$ (written $\rightarrow_{\mathcal{X}'} \preceq \rightarrow_{\mathcal{X}}$) if $Y \rightarrow_{\mathcal{X}'} Z$ implies that $\mathbb{X}\{Y\} \rightarrow_{\mathcal{X}} \mathbb{X}\{Z\}$ for any context $\mathbb{X}$ s.t. $\mathbb{X}\{Y\}$ and $\mathbb{X}\{Z\}$ are well-formed terms of $\mathcal{X}$.*

As examples of embeddings, in our module calculus, $\rightarrow_{\mathcal{T}} \preceq \rightarrow_{\mathcal{C}}$ (because term reductions can be performed in the bindings of a module) and $\rightarrow_{\mathcal{C}} \preceq \rightarrow_{\mathcal{F}}$ (because core module reductions can be performed within a full module term). The self-embedding $\rightarrow_{\mathcal{X}} \preceq \rightarrow_{\mathcal{X}}$ means that the relation $\rightarrow_{\mathcal{X}}$ is a congruence relative to the one-holed contexts of $\mathcal{X}$. For instance, $\rightarrow_{\mathcal{T}}$ and $\rightarrow_{\mathcal{F}}$ are both congruences since they are embedded in themselves.

Together, computational soundness and embedding imply that calculus-based transformations are meaning preserving.

Lemma 1. *If a calculus $\mathcal{X}$ is sound and $\rightarrow_{\mathcal{X}'} \preceq \rightarrow_{\mathcal{X}}$, then any calculus-based transformation T in $\mathcal{X}'$ is meaning preserving in $\mathcal{X}$.*

Proof. By Definition 6, $Y \leftrightarrow_{\mathcal{X}'} T(Y)$ implies that for any context $\mathbb{X}$, $\mathbb{X}\{Y\} \leftrightarrow_{\mathcal{X}} \mathbb{X}\{T(Y)\}$. Then $\mathit{Outcome}_{\mathcal{X}}(\mathbb{X}\{Y\}) = \mathit{Outcome}_{\mathcal{X}}(\mathbb{X}\{T(Y)\})$ by soundness of $\mathcal{X}$. By the definition of $\mathit{Outcome}_{\mathcal{X}}$, $\mathbb{X}\{Y\} \Rightarrow_{\mathcal{X}}^{*} W$ iff $\mathbb{X}\{T(Y)\} \Rightarrow_{\mathcal{X}}^{*} W'$, where W and $W' \in \Rightarrow_{\mathcal{X}}$-NF and $\mathit{Cl}_{\mathcal{X}}(W) = \mathit{Cl}_{\mathcal{X}}(W')$. Since $\mathbf{Value}_{\mathcal{X}}$ respects the ordering of $\mathit{Cl}_{\mathcal{X}}$, W and W' are either both in or both not in $\mathbf{Value}_{\mathcal{X}}$. □

The soundness of the call-by-name and call-by-value λ-calculi are a classic result due to Plotkin [Plo75]. Since the reduction relations of these caculi are congruences (i.e., are self-embedded), Lemma 1 implies that all calculus-based transformations in these calculi are meaning preserving.

A main result of our work is that $\mathcal{T}$, $\mathcal{C}$, and $\mathcal{F}$ are all computationally sound. Given the four embeddings for these calculi enumerated above, Lemma 1 implies that calculus-based transformations are meaning preserving in each of the four cases. Many classic program transformations (both at the term and at the module level) fall into this category: e.g., constant folding and propagation, function inlining, and simple forms of dead-code elimination that eliminate unused value bindings. All of these (and any combinations thereof) can easily be shown to be meaning preserving because all are justified by simple calculus steps.

We emphasize that there are numerous common transformations that are *not* calculus-based and so their meaning preservation cannot be shown via this technique. The operand-swapping SPO transformation introduced above is in this category. Note that $\mathit{Outcome}_{\mathcal{C}}(D_2) = [S \mapsto \mathbf{stuck}(N_1)]$ and $\mathit{Outcome}_{\mathcal{C}}(D_2') = [S \mapsto \mathbf{stuck}(N_2)]$, underscoring that SPO cannot possibly be expressed via calculus steps. Global transformations like closure conversion, assignment conversion, uncurrying, etc., are other examples of non-calculus-based transformations.

3.2 A Novel Technique for Proving Soundness

As in Plotkin's approach, we show soundness of the module calculi in order to prove that calculus-based transformations are meaning preserving. However, we formulate and prove much more general conditions for soundness that do not depend on the particulars of the module calculus or of the definition of a program outcome. We also extend traditionally used definitions to a hierarchy of calculi, allowing terms of one calculus to fill in contexts of another (see Definition 3 above). Our discussion is independent of the particulars of a calculus. The notations M, N for terms and $\mathbb{C}$ for contexts are used below for clarity (since these notations are more traditional); note that they are independent from the same notations used in the term calculus $\mathcal{T}$.

Traditional proofs of computational soundness depend on confluence of reduction in the calculus and on standardization, as well as on the following property, which is often not articulated, but plays a critical role in soundness proofs:

Definition 7 (Class Preservation). *Calculus $\mathcal{X}$ has the class preservation property if $M \hookrightarrow_{\mathcal{X}} N$ implies $Cl_{\mathcal{X}}(M) = Cl_{\mathcal{X}}(N)$, where $M, N \in \mathbf{Term}_{\mathcal{X}}$.*

Below we present a traditional proof of computational soundness that generalizes Plotkin's approach.

Theorem 1 (Soundness of a Confluent Calculus). *Confluence, standardization, and class preservation imply soundness.*

Proof. The diagram of the proof is shown in Fig. 2.[4] Assume that $M \leftrightarrow_{\mathcal{X}} N$ and that $M \Rightarrow^* M' = \mathit{Eval}(M)$. By confluence there exists L s.t. $M' \rightarrow^* L$, $N \rightarrow^* L$. Since M' is a normal form w.r.t. $\Rightarrow$, there can not be an evaluation sequence starting at M', so $M' \hookrightarrow^* L$. By standardization, $N \rightarrow^* L$ implies that there is N' s.t. $N \Rightarrow^* N' \hookrightarrow^* L$. Since M', L, and N' are connected only by $\hookrightarrow$, by class preservation, $\mathit{Cl}(M') = \mathit{Cl}(L) = \mathit{Cl}(N')$, and since N' is of the same class as M', it must also be a normal form w.r.t. $\Rightarrow$, so $N' = \mathit{Eval}(N)$.

Now assume that M diverges. If $\mathit{Eval}(N)$ exists, then by the above argument we can show that $\mathit{Eval}(M)$ exists as well. So if M diverges, then so does N. □

The above approach does not work for a calculus that lacks confluence. But it turns out that general confluence is not required for soundness! Since the outcome of a term is defined via the evaluation reduction, we can instead use a weaker form of confluence: *confluence with respect to evaluation.* The two properties given below that we call *lift* and *project* (see also Fig. 3), together with the class preservation property, are sufficient to show soundness.

Definition 8 (Lift). *A calculus has the lift property if for any reduction sequence $M \hookrightarrow N \Rightarrow^* N'$ there exists a sequence $M \Rightarrow^* M' \hookrightarrow^* N'$.*

[4] In Figs. 2 and 3, double-headed arrows denote reflexive, transitive closures of the respective relations, and a line with arrows on both ends denotes the reflexive, symmetric, transitive closure of the respective relation.

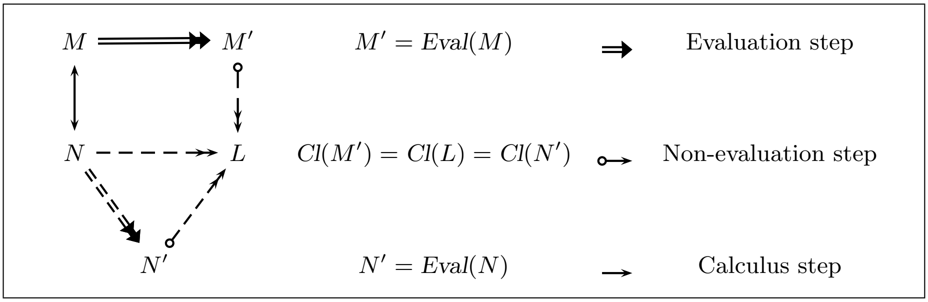


Fig. 2. Sketch of the traditional proof of computational soundness.

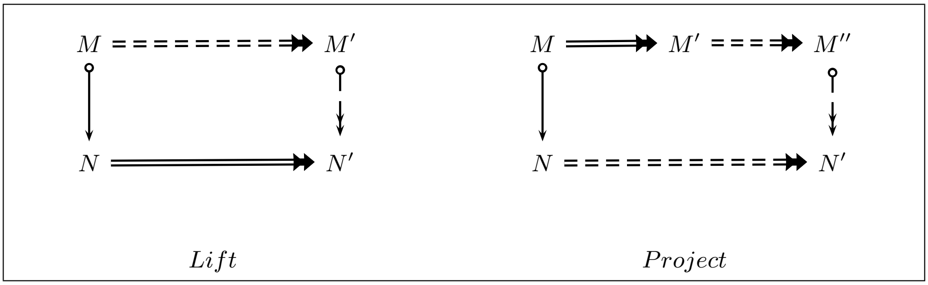


Fig. 3. The lift and project properties.

Definition 9 (Project). *A calculus has the project property if $M \hookrightarrow N$, $M \Rightarrow^* M'$ implies that there exist terms M'', N' s.t. $M' \Rightarrow^* M''$, $N \Rightarrow^* N'$, and $M'' \hookrightarrow^* N'$.*

The project property is the formalization of the notion of confluence w.r.t. evaluation mentioned above. It says that an evaluation step and a non-evaluation step leaving the same term can always be brought back together. The lift property is equivalent to standardization: any reduction sequence can be transformed into a standard sequence by pushing "backwards" sequences of evaluation steps through single non-evaluation steps. There is a benefit in proving standardization using the lift property (rather than directly): proofs of both the lift and project properties use the same mechanism (certain properties of residuals and finite developments [Bar84]) and share several intermediate results.

The following theorem embodies our new approach to proving soundness:

Theorem 2. *Suppose that $\Rightarrow$ is confluent. Then lift, project, and class preservation imply soundness.*

Proof. We want to show that if $M \leftrightarrow N$, then $\mathit{Outcome}(M) = \mathit{Outcome}(N)$. Without loss of generality assume that M and N are connected by a single step.

Assume that $\mathit{Outcome}(M) \neq \bot$. Let $M' = \mathit{Eval}(M)$. In all four of the following cases, $\mathit{Outcome}(M) = \mathit{Outcome}(N)$:

- $M \hookrightarrow N$. By the project property, $M \Rightarrow^* M'$ implies that there exist M'', N' s.t. $M' \Rightarrow^* M''$, $N \Rightarrow^* N'$, and $M'' \hookrightarrow^* N'$. But M' is a normal form w.r.t. $\Rightarrow$, so $M' = M''$. By the class preservation property $\mathit{Cl}(M') = \mathit{Cl}(N')$, so N' is also a normal form. Hence $N' = \mathit{Eval}(N)$, and $\mathit{Outcome}(M) = \mathit{Outcome}(N)$.
- $N \hookrightarrow M$. Similar to the previous case by the lift property.
- $M \Rightarrow N$. By confluence of $\Rightarrow$ there exists N' s.t. $N \Rightarrow^* N'$, $M' \Rightarrow^* N'$. But M' is a normal form, so $N \Rightarrow^* M' = \mathit{Eval}(N)$.
- $N \Rightarrow M$. Then by transitivity of $\Rightarrow^*$, $N \Rightarrow^* M' = \mathit{Eval}(N)$.

Now let $\mathit{Outcome}(M) = \bot$. Assuming $\mathit{Outcome}(N) \neq \bot$, by the above argument $\mathit{Outcome}(M) = \mathit{Outcome}(N) \neq \bot$, and we get a contradiction. □

$\mathcal{C}$ and $\mathcal{F}$ satisfy the lift, project, and class preservation properties, so they enjoy the soundness property. For the technical details, consult [MT00].

4 Weak Distributivity

We say that a module transformation T is weakly distributive if and only if $T(D_1 \oplus D_2) = T(T(D_1) \oplus T(D_2))$, where $=$ is syntactic equality (modulo α-renaming and module binding order).

Let T_{link} be a single module transformation performing all link-time optimizations. Suppose that the translator from source modules to intermediate modules is given by $\mathit{s2i}(D) = T_{\text{link}}(D)$[5]. Also suppose that the linking operator on intermediate modules is defined as $D_1 \oplus_{\text{link}} D_2 = T_{\text{link}}(D_1 \oplus D_2)$. Then if T_{link} is weakly distributive, we have that $\mathit{s2i}(D_1) \oplus_{\text{link}} \mathit{s2i}(D_2) = T_{\text{link}}(T_{\text{link}}(D_1) \oplus T_{\text{link}}(D_2))$ $= T_{\text{link}}(D_1 \oplus D_2) = \mathit{s2i}(D_1 \oplus D_2)$. Thus, compiling a "link tree" of modules in the link-time compilation model gives exactly the same code as compilation in whole-program model. This is the sense in which weakly distributive transformations are promising candidates for link-time optimizations.

Here we briefly discuss two classes of weakly distributive module transformations T. We assume the following about T: (1) it is strongly normalizing; and (2), if T can be applied to a module $[X_i \overset{n}{\underset{i=1}{\mapsto}} M_i]$, then it can be applied to a module $[X_i \overset{n}{\underset{i=1}{\mapsto}} M_i, Y_j \overset{m}{\underset{j=1}{\mapsto}} N_j]$, i.e. to the same module with extra bindings. To motivate the second assumption, let FI be function inlining on modules restricted to non-recursive substitutions (so that the first assumption is satisfied). Consider the following inlining/linking sequence: $[X \mapsto \lambda w.Y, Z \mapsto \lambda x.X] \oplus [Y \mapsto \lambda y.Z] \overset{\text{FI}}{\to} [X \mapsto \lambda w.Y, Z \mapsto \lambda x.\lambda w'.Y] \oplus [Y \mapsto \lambda y.Z] \to_{\mathcal{F}} [X \mapsto \lambda w.Y, Z \mapsto \lambda x.\lambda w'.Y, Y \mapsto \lambda y.Z] \overset{\text{FI}}{\to} [X \mapsto \lambda w.\lambda y.Z, Z \mapsto \lambda x.\lambda w'.Y, Y \mapsto \lambda y.Z]$. On the other hand, linking first gives: $[X \mapsto \lambda w.Y, Z \mapsto \lambda x.X, Y \mapsto \lambda y.Z]$, and at this point the cycle becomes apparent, and no inlining is possible. Thus, extra bindings can prevent weak distributivity by blocking the transformation.

[5] For simplicity, we assume the source and intermediate languages are the same.

A simple class of weakly distributive transformations are those satisfying two conditions: (1) idempotence: $T(T(D)) = T(D)$; and (2) (strong) distributivity over $\oplus$: $T(D_1 \oplus D_2) = T(D_1) \oplus T(D_2)$. It is easy to show that such a T is weakly distributive. Examples include many combinations of intra-term transformations, such as constant folding/propagation, dead code elimination, and function inlining (restricted to non-recursive cases). Note that the second condition implies that the transformation independently transforms the components of a module; i.e., the transformation cannot use the (subst) or (GC) rule.

Closures of confluent transformations T form another class of weakly distributive transformations. It is possible to simulate any transformation step in $T(T(D_1) \oplus T(D_2))$ by a corresponding step in $T(D_1 \oplus D_2)$. Using confluence, strong normalization, and the extra-bindings assumption, it can be shown that the two expressions transform to the same result. For example, constant folding/propagation at the module level (i.e., including the (subst) rule) has all of these properties, and so is weakly distributive.

5 Future Work

There are several directions in which we plan to extend the work presented here.

Types: We are exploring several type systems for our module calculus, especially ones which express polymorphism via intersection and union types. These have intriguing properties for modular analysis and link-time compilation [Jim96,Ban97,KW99].

Non-local Transformations: So far, we have only considered meaning preservation and weak distributivity in the context of simple local transformations. We are investigating global transformations like closure conversion, uncurrying, and useless variable elimination in the context of link-time compilation.

Weakening Weak Distributivity: Weak distributivity requires the rather strong condition of synactic equality between $T(D_1 \oplus D_2)$ and $T(T(D_1) \oplus T(D_2))$. Weaker notions of equality may also be suitable. Note that "has the same meaning as" is *too* weak, since it does not capture the pragmatic relationship between the two sides; they should have "about the same efficiency".

Abstracting over the Base Language: Our framework assumes that the module calculus is built upon a particular base calculus. Inspired by [AZ99], we would like to parameterize our module calculus over any base calculus.

Pragmatics: We plan to empirically evaluate if link-time compilation can give reasonable "bang for the buck" in the context of a simple prototype compiler.

References

AB97. Z. M. Ariola and S. Blom. Cyclic lambda calculi. In *TACS 97, Sendai, Japan*, 1997.

AF97. Z. M. Ariola and M. Felleisen. The call-by-need lambda calculus. *J. Funct. Prog.*, 3(7), May 1997.

AFM$^+$95. Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. In *Conf. Rec. 22nd Ann. ACM Symp. Princ. of Prog. Langs.*, pp. 233–246, 1995.

AK97. Z. M. Ariola and J. W. Klop. Lambda calculus with explicit recursion. *Inf. & Comput.*, 139(2):154–233, 15 Dec. 1997.

AZ99. D. Ancona and E. Zucca. A primitive calculus for module systems. In G. Nadathur, ed., *Proc. Int'l Conf. on Principles and Practice of Declarative Programming*, LNCS, Paris, France, 29 Sept. – 1 Oct. 1999. Springer-Verlag.

Ban97. A. Banerjee. A modular, polyvariant, and type-based closure analysis. In *Proc. 1997 Int'l Conf. Functional Programming*, 1997.

Bar84. H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, revised edition, 1984.

Car97. L. Cardelli. Program fragments, linking, and modularization. In POPL '97 [POPL97].

CHP97. K. Crary, R. Harper, and S. Puri. What is a recursive module? In *Proc. ACM SIGPLAN '97 Conf. Prog. Lang. Design & Impl.*, 1997.

DEW99. S. Dossopoulou, S. Eisenbach, and D. Wragg. A fragment calculus – towards a model of separate compilation, linking, and binary compatibility. In *Proc. 14th Ann. IEEE Symp. Logic in Computer Sci.*, July 1999.

DS96. D. Duggan and C. Sourelis. Mixin modules. In *Proc. 1996 Int'l Conf. Functional Programming*, pp. 262–273, 1996.

Fer95. M. F. Fernandez. Simple and effective link-time optimization of Modula-3 programs. In *Proc. ACM SIGPLAN '95 Conf. Prog. Lang. Design & Impl.*, pp. 103–115, 1995.

FF98. M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *Proc. ACM SIGPLAN '98 Conf. Prog. Lang. Design & Impl.*, 1998.

FH92. M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comp. Sc.*, 102:235–271, 1992.

HL94. R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In POPL '94 [POPL94], pp. 123–137.

Jim96. T. Jim. What are principal typings and what are they good for? In *Conf. Rec. POPL '96: 23rd ACM Symp. Princ. of Prog. Langs.*, 1996.

KW99. A. J. Kfoury and J. B. Wells. Principality and decidable type inference for finite-rank intersection types. In *Conf. Rec. POPL '99: 26th ACM Symp. Princ. of Prog. Langs.*, pp. 161–174, 1999.

Ler94. X. Leroy. Manifest types, modules, and separate compilation. In POPL '94 [POPL94], pp. 109–122.

Ler96. X. Leroy. A modular module system. Tech. Rep. 2866, INRIA, Apr. 1996.

MT00. E. Machkasova and F. Turbak. A calculus for link-time compilation. Technical report, Comp. Sci. Dept., Boston Univ., 2000.

PC97. M. P. Plezbert and R. K. Cytron. Is "just in time" = "better late than never"? In POPL '97 [POPL97], pp. 120–131.

Plo75. G. D. Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theor. Comp. Sc.*, 1:125–159, 1975.

POPL94. *Conf. Rec. 21st Ann. ACM Symp. Princ. of Prog. Langs.*, 1994.

POPL97. *Conf. Rec. POPL '97: 24th ACM Symp. Princ. of Prog. Langs.*, 1997.

SA93. Z. Shao and A. Appel. Smartest recompilation. In *Conf. Rec. 20th Ann. ACM Symp. Princ. of Prog. Langs.*, 1993.

WV99. J. B. Wells and R. Vestergaard. Confluent equational reasoning for linking with first-class primitive modules (long version). Full paper with three appendices for proofs, Aug. 1999.

Improving the Representation of Infinite Trees to Deal with Sets of Trees

Laurent Mauborgne

LIENS – DMI, École Normale Supérieure, 45 rue d'Ulm, 75 230 Paris cedex 05, France
Tel: +33 (0) 1 44 32 20 66; Email: Laurent.Mauborgne@ens.fr
WWW home page: http://www.dmi.ens.fr/~mauborgn/

Abstract. In order to deal efficiently with infinite regular trees (or other pointed graph structures), we give new algorithms to store such structures. The trees are stored in such a way that their representation is unique and shares as much as possible. This maximal sharing allows substantial memory gain and speed up. For example, equality testing becomes constant time. The algorithms are incremental, and as such allow good reactive behavior. This new algorithms are then applied to the representation of sets of trees. The expressive power of this new representation is exactly what is needed by set-based analysis.

1 Introduction

When applying set-based analysis techniques for practical applications, one is surprised to see that the representation of the sets of trees is not very efficient. Even when we use tree automata, we cannot overcome this problem without performing a minimization of the whole automaton at each step. We propose a new way of dealing with this kind of structure to get a representation that is as small as possible during the computation.

After analysis of the problem, it appears that the underlying structure we want to optimize can be described mathematically as regular infinite trees. Because tree structures appear everywhere in computer science where a hierarchy occurs, we found it interesting to present the algorithms in an independent way. In this way, our technique appears as an extension of an efficient solution to store finite trees.

The representation we extend uses just the minimum amount of memory by sharing equivalent subtrees. This saves a lot of space. It is used, for example, with sets of words represented as a tree to share common prefixes. It is possible to share the subtrees incrementally, and at the same time to give a unique representation to different versions of the same trees. Such a technique allows constant time equality testing and a great speed up for many other algorithms manipulating trees. It has been the source of the success of Binary Decision Diagrams (BDDs) [2], which are considered the best representation for boolean functions so far.

But as soon as a loop occurs somewhere in the data, finite tree techniques are no longer adequate. The main contribution of this article is to extend the good

G. Smolka (Ed.): ESOP/ETAPS 2000, LNCS 1782, pp. 275–289, 2000.

results of unique sharing representation from finite trees to infinite trees. These techniques are applied to the representation of sets of trees in set-based analysis, but they can also be applied directly to the representation and manipulation of finite automata, or infinite boolean functions [14].

After a recollection of the classic results over finite trees in section 2, we present the solutions for the most difficult problems with infinite trees in the section 3 on cycles. The general problem is then treated in section 4, with a full example. Complexity issues and algorithms to manipulate infinite trees are discussed in section 5. The application to sets of trees implies the description of a new encoding to keep the uniqueness of the representation. This new contribution is described in section 6.

2 Classic Representation of Trees

2.1 Trees and Graphs

As we deal with the computer representation of data structures, we must give a clear meaning to the word representation, and in particular clearly distinguish between what is represented and what is the representation. For this reason, we will give a mathematical definition of what is a tree, and another one for the way it is usually stored in a computer.

Let $\mathbb{N}^*$ be the set of words over $\mathbb{N}$, ε denoting the empty word. We note $\prec$ the prefix ordering on words and $u.v$ the concatenation of the words u and v. Let F be a finite set of labels.

Definition 1. *A* tree *t labeled by F is a function of $pos(t) \to F$ such that $pos(t) \subset \mathbb{N}^*$ and $\forall p \in \mathbb{N}^*, \forall i \in \mathbb{N}$, $p.i \in pos(t) \Rightarrow (p \in pos(t)$ and $\forall j < i$, $p.j \in pos(t))$*

Let $p \in pos(t)$. The subtree of t in p, written $t_{[p]}$ is defined by: $pos(t_{[p]}) \overset{\mathrm{def}}{=} \{q \in \mathbb{N}^* \mid p.q \in pos(t)\}$, and $t_{[p]}(q) \overset{\mathrm{def}}{=} t(p.q)$. A tree is uniquely determined by the label of its root, $t(\varepsilon)$, and by the children of the root, the different $t_{[i]}, i \in \mathbb{N}$. In the sequel, a generic tree will be denoted $\begin{array}{c} f \\ \swarrow\ \searrow \\ t_0 \cdots t_{n-1} \end{array}$, where f is the label of the root, and $(t_i)_{i<n}$ are the children of the root.

When representing a tree in a computer, we usually use one computer location for each position p in $pos(t)$, where we store the label $t(p)$ and the location of the different children (the $p.i$'s in $pos(p)$) of this position. Such a representation is well modeled by a graph, where each node of the graph corresponds to a computer location. We do not give the most general definition of graphs, but the definition that is useful in this article to represent trees.

Definition 2. *A* graph *G labeled by F is composed of two sets, the node set, G^N, and the edge set, $G^E \subset G^N \times G^N \times \mathbb{N}$, and every node of the graph is associated with a label in F.*

We define the notion of path in a graph: let $p \in \mathbb{N}^*$, p is a *path* of the node N if and only if $p = \varepsilon$ or $p = i.q$ and there is an $M \in G^N$ such that $(N, M, i) \in G^E$ and q is a path of M. If O is the only node at the end of the path, we write $N.p = O$. We define $\mathcal{G}(N)$ as the graph defined by the modes which can be reached from N. We will often identify a node N and the graph $\mathcal{G}(N)$.

Definition 3. *A node N* represents *a tree t if and only if the set of paths of N is $pos(t)$, and $\forall p \in pos(t)$, $N.p$ is well defined, and its label is $t(p)$.*

A *finite tree* t is a tree such that $pos(t)$ is finite. There is always a possible representation by a finite graph for finite trees. In the most common use, one node corresponds to each path of the finite tree.

A *regular* tree t is a tree such that the number of distinct subtrees of t is finite. Such a tree can be infinite, but it can still be represented by a finite graph [6], see Fig. 1 for an example.

$$t = \begin{cases} (10)^* & \to f \\ (10)^*0 & \to a \\ 1(01)^* & \to g \end{cases} \qquad \text{can be represented by}$$

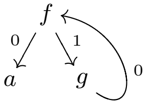


Fig. 1. An infinite regular tree

2.2 Best Representation

The naive representation, which consists in using any graph representing the tree [6], is very easy to deal with and quite widely used for small problems. But we can do far better if we observe that some nodes can represent different paths of the tree, as long as the subtrees at these paths are the same. This is called sharing the subtrees (see e.g. [1]). In fact, the best we can do is to have exactly one node for each distinct subtree. This is what we call the best representation of a tree. In the case of finite trees, this can save a lot of space, and even time by memoizing [15], and in the case of infinite regular trees, we avoid the possibility of unbounded representation for a given tree.

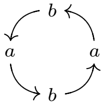
When dealing with many trees, we can do even better: considering the entire computer memory as one graph, we can optimize the representation for all the trees, and have in effect exactly one memory location for each distinct tree we need to store. An immediate consequence is that we just have to compare the location of the roots (the node representing the trees) to compare entire trees. Such a technique is used e.g. in BDDs [2] to achieve impressive speed up and memory gain.

The technique to obtain the best representation of the trees uses a dictionary mechanism linking keys to nodes of the graph, usually a hash table. The keys are built incrementally: if the keys for the $(t_i)_{i<n}$ are known and linked to the

nodes $(N_i)_{i<n}$, then the key for $\begin{smallmatrix} & f & \\ \swarrow & & \searrow \\ t_0 & \cdots & t_{n-1} \end{smallmatrix}$ is $(f, (N_i)_{i<n})$. Each time a key is not present in the dictionary, it is associated with a new node N, with edges to the N_i's. If we come to a tree whose key is already in the dictionary, we use the corresponding node. As the trees are always built from leaves to root, we have indeed a best representation for the trees.

3 Dealing with Cycles

When representing infinite trees, though, we cannot go from the leaves to the root, so we cannot start the key mechanism which leads to the best representation. The difficulty lies in the infinite paths of the tree, that is the cycles of the graph representing the tree. Whereas in finite trees there is no need to see beyond the immediate children of a given node, when dealing with cycles, we can have reasons to look further, in order to detect the two causes of cycle unfolding: cycle growth and root unfolding. For example, consider the cycle $a \rightleftarrows b$.



is an example of cycle growth, and $a \to b \rightleftarrows a$ is an example of root unfolding. In this very simple example, it is easy to reduce root unfolding by looking at the key of the root, but it is much more difficult if the root itself is still in another cycle. In order to concentrate on the real difficulties, we suppose in this section that we deal with strongly connected graphs, that is graphs such that there is a path between any pair of nodes.

3.1 Cycle Growth and Tree Keys

We give $\equiv_{\text{tree}}$ as the equivalence between nodes representing the same tree. The goal of cycle growth reduction is to find an equivalent graph with the minimum number of nodes. In such a graph, whatever the nodes N and M, $N \equiv_{\text{tree}} M \Rightarrow N = M$. Such a problem is called a partitioning problem. It has been solved in time $n\log(n)$ by Hopcroft [10] for finite automata, and in the general case by [4]. We call `share`(N) the algorithm that takes a node N and modifies the associated graph so that it has the fewest possible nodes (Fig. 2).

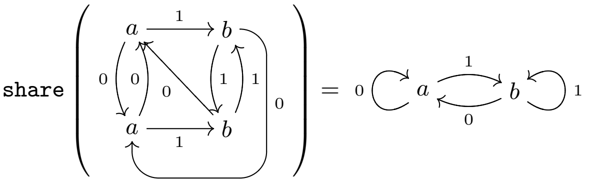


Fig. 2. Application of the `share` algorithm.

Cycle growth reduction corresponds to the state of the art in automata representation. But we want to go further: we need that the representation be unique whatever the different versions of the same tree. To perform this, we give a key which distinguishes between non isomorphic graphs. This key is associated to a given node N of the graph. It is a finite tree which corresponds to the graph as long as we do not loop, but as soon as we loop, the label of the node is replaced by its access path from N. It is described as $\texttt{treeKey}(N)$. See Fig. 3 for an example. The isomorphism between graphs is not the same thing as $\equiv_{\text{tree}}$.

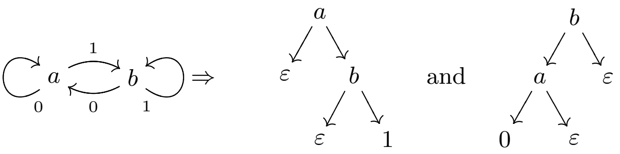


Fig. 3. A graph, followed by the tree keys of its two nodes

In general it can differentiate two graphs which represent the same tree. The interesting point is that it is indeed the same relation on graphs with a minimal number of nodes.

Proposition 1. *Whatever M and N, such that $\mathcal{G}(M)$ and $\mathcal{G}(N)$ are graphs with minimal number of nodes, $\texttt{treeKey}(M) = \texttt{treeKey}(N) \Leftrightarrow M \equiv_{\text{tree}} N$.*

Proof. The difficult point is $M \equiv_{\text{tree}} N \Rightarrow \texttt{treeKey}(M) = \texttt{treeKey}(N)$. Suppose there are M and N such that $\mathcal{G}(M)$ and $\mathcal{G}(N)$ are graphs with minimal number of nodes, $M \equiv_{\text{tree}} N$ and $\texttt{treeKey}(M) \neq \texttt{treeKey}(N)$. Let $t_M = \texttt{treeKey}(M)$ and $t_N = \texttt{treeKey}(N)$. Because $t_M \neq t_N$, there is a path p such that $t_M(p) \neq t_N(p)$. But if $t_M(p)$ is a label of the graph, $t_M(p)$ is the label of $M.p$, and the same holds for N. Because $M \equiv_{\text{tree}} N$, $M.p$ and $N.p$ have the same label, so at least one of $t_M(p)$ or $t_N(p)$ is not a label of the graphs (and so is in $\mathbb{N}^*$), say $t_M(p)$. It means there is a $q \prec p$ such that $M.q \equiv_{\text{tree}} M.p$. So $N.q \equiv_{\text{tree}} N.p$, but by minimality of the number of nodes of $\mathcal{G}(N)$, $N.q$ and $N.p$ must be the same node, and so $t_N(p) = q = t_M(p)$. □

Because we can find an equivalent graph with minimal number of nodes for strongly connected graphs, we have a valid key mechanism for any strongly connected graph: we first apply `share`, then `treeKey`.

3.2 Root Unfolding and Partial Keys

With just `share` and `treeKey` (applied to every node), we can have a unique representation that shares common subtrees. But as we need to start the whole process from the beginning for each little modification in the trees, such a process would be quite slow. Moreover, it is much better to apply the `share` algorithm on the smallest possible graphs. As it is not a linear algorithm, we have better results if we can split the graph and apply the algorithm to each separate subgraph only.

The finite parts of the tree can always be treated in the classic way, while the loops will need a special treatment. In order to decompose the graph and mark those parts of the graph which have been definitely treated, we introduce partial keys. A partial key looks like a node key for a finite tree, a label followed by a vector of nodes, except that for some parts of the vector, there is no node (see Sect 4.3 for an example). A partial key k has a name: $\text{name}(k) \in F$ and is a partial function from $\mathbb{N}$ to nodes. A graph labeled by partial keys is such that for every node N in the graph, if k is the partial key for N, the edges in the graph correspond to those integers for which the partial key is not defined. For example, if a node is labeled by f of arity 3, we can have a partial key which is not defined on 0 and 1 (we write a $\bullet$), and on 2 its value is the node number 4. We write $(f, \bullet\bullet\Box_4)$ for this partial key. The only edges that can leave from such a node would be labeled by 0 and 1. The idea is that what is in the partial keys is uniquely represented. In our example, the node number 4, $\Box_4$, is a unique representation of some tree. Later on during the computation, it is possible that we have a unique representation for the first component, say with node $\Box_2$, and the partial key becomes $(f, \Box_2 \bullet \Box_4)$. When a partial key is full (defined everywhere), then the node should be a unique representation.

This new graphs have new equivalence relation, $\equiv_{\text{pk}}$ which is implied by $\equiv_{\text{tree}}$. This new equivalence relation corresponds to $\equiv_{\text{tree}}$ after the expansion of the partial keys into the graph.

But now, with those partial keys, we can have a strongly connected graph such that, by root unfolding, one of its nodes is equivalent to a node in a partial key. Figure 4 shows a case of root unfolding, which can be as big as we want, even after cycle growth reduction[1]. So, we must look for such a node, even before

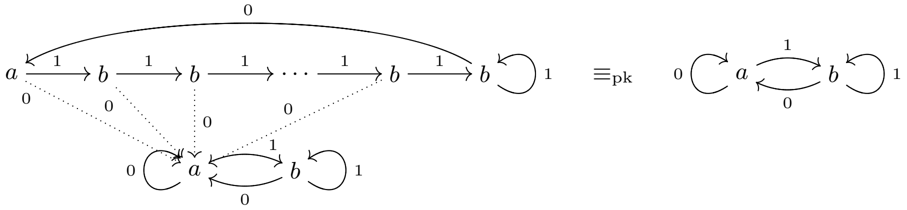


Fig. 4. Root unfolding of a cycle

applying the `share` algorithm.

The name of the algorithm performing this task is `shareWithDone`(N). It returns N if and only if no other node in the partial keys is equivalent to N. Otherwise, it returns the node in the partial keys that is equivalent to N. This

[1] In this figure, dotted lines correspond to nodes stored in partial keys.

algorithm uses some properties of the graph to reduce the complexity of the computation. Let G be the graph associated with N. As always in this section, we suppose that G is strongly connected. We call H the graph already computed and that is reachable from the partial keys of G. The algorithm determines whether a node of G is equivalent to a node of H. If it is the case, then there is root unfolding. If not, there is no root unfolding. We show that it is enough to verify this property for one node to treat the entire graph G because G is strongly connected. Suppose N is equivalent to M in H. Then, whatever the legible path p, $N.p$ is equivalent to $M.p$. Because H has been treated already, any $M.p$ is in H, and because G is strongly connected, any node of G is a $N.p$.

There is a kind of reciprocal property that is exploited too: for some subsets of H^N, if no node of the subset is equivalent to a particular node of G, then they are not equivalent to any node of G. A subset of H^N is said to be closed if and only if, for every legible path p, for every node N in the subset, $N.p$ is in the subset.

Proposition 2. *$\forall H' \subset H^N$ such that H' is closed, if $\exists N \in G^N$ such that $\forall M \in H'$, $N \not\equiv_{\text{pk}} M$, then this holds for every $N \in G^N$.*

Proof. Let H' be such a subset and N a node of G. If N is not equivalent to any node in H', then, suppose there is a $M \in G^N$ and a $O \in H'$ such that M is equivalent to O. As G is strongly connected, there is a p such that$M.p = N$. So, N would be equivalent to $O.p$, which is in H'. This proves that no element of G^N is equivalent to any element of H'. □

Because of these properties, we can use the following algorithm for `shareWithDone`: we just compare every nodes of G with the nodes that are reachable from their partial keys and not already encountered. This comparison can be quite efficient by exploiting the fact that the nodes in the partial keys are unique representations of trees, although we have a quadratic worst case complexity.

We will show in the next section, that by applying first `shareWithDone`, then `share` and then `treeKey`, we can indeed represent uniquely (and with the least possible number of nodes) any strongly connected graph, in an incremental process.

4 The Best Representation for Infinite Trees

4.1 Informal Presentation

In order to show how we can produce the best representation for an infinite tree, we solve the following problem: considering a graph representing a tree t, return an equivalent graph with a minimal number of nodes. To achieve this in an incremental way, we use two dictionary mechanisms and a decomposition of the graph. First, we apply the classic algorithm, using the dictionary D, on the finite subtrees of the tree. When a finite subtree is entirely treated, it is incorporated in the graph through partial keys. Second, when there is no more finite subtree, there is a subtree represented by a strongly connected graph. The dictionary D_G stores the tree keys of such graphs, and after shareWithDone and if necessary, share, we can decide whether another equivalent graph has already been encountered, and if not, use new nodes. When the strongly connected graph is treated, it is considered as just a node, and so we can iterate on our algorithm until we give the representation of the root.

4.2 The Algorithm

We suppose given a dictionary D which maps full keys to nodes corresponding to a unique representation of the associated tree, and a dictionary D_G which maps tree keys (in fact keys of these finite trees) to nodes corresponding to a unique representation of the associated strongly connected graph.

The algorithm uses local dictionaries too, which we assume to be empty when the process starts on a tree. The dictionary encountered contains the nodes of the original representation already encountered (so that we do not loop). The set returnNodes is used to detect the roots of the loops.

A node is considered "treated" when it is in the dictionary D (and so it represents uniquely a tree). To decide whether a node is "treated", we just have to look at its key: it is "treated" if the key is full.

```
representation($t$)
  Step 1  if $t \in$ encountered then
                if encountered($t$) is not treated add it in returnNodes
                return encountered($t$)
  Step 2  $N$ is a new node labeled by the empty partial key $k$ of name
                      the label of $t$
  Step 3  for each child $t_i$ of $t$ do
          3a    $N_i \leftarrow$ representation($t_i$)
          3b    if $N_i$ is treated, then add it to $k$
                else $N.i \leftarrow N_i$
  Step 4  if $k$ is full then
                if $k \in D$ return $D(k)$
                else add $k \to N$ to $D$ and return $N$
  Step 5  remove $N$ from returnNodes
  Step 6  if returnNodes $= \emptyset$ then return representCycle($N$)
  Step 7  return $N$
```

```
representCycle($N$)
  Step 1  if shareWithDone($N$) $\neq N$ then return shareWithDone($N$)
  Step 2  share($N$)
  Step 3  if treeKey($N$) $\in D_G$ then return $D_G$(treeKey($N$))
  Step 4  for each node $M$ in the graph defined by $N$ do
          4a    add treeKey($M$)$\to M$ to $D_G$
          4b    add the children of $M$ to its partial key $m$
          4c    add $m \to M$ to $D$
  Step 5  return $N$
```

4.3 Example

We present the algorithm to represent regular trees on an example, the graph of Fig. 5, where each node is assigned a number. We will write t_i for the tree represented by the node number i, $\Box_i$.

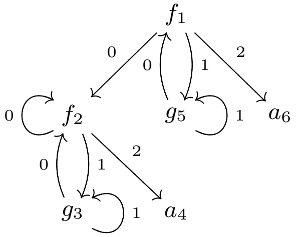
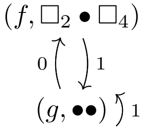
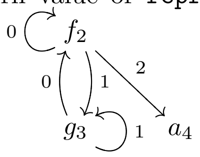


Fig. 5. Example

`representation`(t_1) calls `representation` for t_2, t_5 and t_6. The call to `representation` on t_2 will return the node $\Box_2$. It will also store various nodes in D, and in particular $(a) \to \Box_4$. The call on t_5 will just return an untreated node $\Box_5$, with nothing added in the dictionaries. The call on t_6 will recognize on step 4 that a is in D and so it will return $\Box_4$.

Thus, at step 5, `returnNodes` $= \{\Box_1\}$ becomes empty, and we call `representCycle` with the graph[2] 

. A call to `shareWithDone` returns the node $\Box_2$. So the return value of `representation` on t_1 is $\Box_2$, the node labeled by f in the graph 

. Moreover, the dictionaries will be:

$$D = \{(a) \to \Box_4, (g, \Box_3\Box_2) \to \Box_3, (f, \Box_2\Box_3\Box_4) \to \Box_2\}$$

$$D_G = \left\{ \begin{tikzcd} & (f, \bullet\bullet\Box_4) \arrow[dl] \arrow[d] & & & (g, \bullet\bullet) \arrow[d] \arrow[dr] & \\ \varepsilon & (g, \bullet\bullet) \arrow[dl] \arrow[dr] & \to \Box_2, & & (f, \bullet\bullet\Box_4) \arrow[dl] \arrow[dr] & \varepsilon \to \Box_3 \\ \varepsilon & & 1 & 1 & & \varepsilon \end{tikzcd} \right\}$$

[2] Remember that $(f, \Box_2 \bullet \Box_4)$ is the partial key which is not defined on its second component.

4.4 Proof of the Algorithm

The algorithm returns the node of a graph. We must prove that this graph represents the same tree as the original graph, and that it is a graph of maximal sharing.

First, notice that the algorithm terminates, because of the dictionary `encountered` which implies that each node of the original graph is treated only once.

The correctness of the algorithm is derived from the fact that we return the same graph as the original, except when we recognize that an equivalent node had already been encountered (through the node keys or the tree keys), in which case we replace one node by the other. It is the case step 4 of `representation`, and steps 1, 2 and 3 of `representCycle`

The fact that the resulting graph has the minimal number of nodes lies in the use of the dictionaries D and D_G to ensure that we never duplicate any node. The dictionary D contains the node keys of every node encountered, and the dictionary D_G contains the tree key of every node of every strongly connected graph with minimal number of nodes we encounter. We can prove that each time we definitely introduce new nodes, there is no duplication. Definitive introduction is performed in two points: step 4 of `representation`, and step 4 of `representCycle`.

Step 4 of `representation`, we know that the key k is not in D. Moreover, each one of the N_i composing the key is unique because nodes in partial keys have already been treated. So if a tree $\begin{array}{c} f \\ \swarrow \ \searrow \\ t_0 \cdots t_{n-1} \end{array}$ had already been encountered, the key $(f, (N_i)_{i<n})$ would already have been encountered.

Step 4 of `representCycle`, we know that the key `treeKey`(`share`(N)) has never been encountered before. Because such a key is valid for strongly connected graphs, it means that no other node M such that $M \equiv_{\text{tree}} N$ have been encountered before. But the problem is that we have a partial key semantics on these graphs, and $\equiv_{\text{tree}} \subset \equiv_{\text{pk}}$, so we could have $M \not\equiv_{\text{tree}} N$ but $M \equiv_{\text{pk}} N$ in effect representing the same tree. Because $M \not\equiv_{\text{tree}} N$, there is a path p such that $M.p$ and $N.p$ do not have the same label, k_M and k_N. But as N and M represent the same tree, k_M and k_N must have the same name, so their only possible difference is in the partial function. It means there is an i such that one of the keys is defined on i and not the other key (if both of them were defined on i, their value would be the same on i, as the nodes in partial keys are unique representations). By construction, the nodes M and N are in strongly connected graphs. So if one of the keys is not defined on i, there is a q such that $M.piq = M$ or $N.piq = N$. If t is the tree represented by both nodes, it means that $t_{[piq]} = t$. Suppose k_M is defined on i, then there is a node reachable from $k_M(i)$ which represents the same tree as M, and as such it would have been found by `shareWithDone`. So the graph defined by M would never have gone beyond the step 1 of `representCycle`. It means that another representative is stored for the cycle (we go on like this until we find one which is equivalent to N, which means that the test step 3 could not have been false). If k_N is defined on i, by the same argument, we could not have been beyond the step 1, and so no new node is created.

If no node equivalent to N has been encountered, it is the same for every other node M in the graph represented by N. It is due to the strong connectivity of the graph which implies that if M has already been encountered, N has already been encountered.

5 Complexity Issues

Algorithms on shared trees can be more difficult than standard algorithms on trees, because we must keep the uniqueness of the representation, and for efficiency, we must do it incrementally. Comparing complexities of algorithms on the two representations (the naive and the sharing ones) is difficult, though. The complexity is measured with respect to the size of the inputs of the algorithms, which can be reduced to the number of nodes of the inputs in our case. In the case of shared regular trees, the number of nodes is exactly the number of distinct subtrees of the tree, but when the tree is not shared, the number of nodes can be of any value greater than the number of distinct subtrees. In the sequel, we denote by n this number of nodes, but we must keep in mind that this n can be much bigger in the case of non-shared trees.

The basic property of shared trees is the uniqueness of the representation. Thus, testing tree equality is really immediate: we just compare the memory location of the root. In the classic case, the best method uses a partitioning algorithm. Another case where we can avoid such a computation with shared trees is testing if a tree is a subtree of another one. In the shared case, we just have to compare the root of the first tree with all the nodes of the second one. Not only is it linear, but the second tree is very likely to have very less nodes in the shared case than in the classic representation.

When building finite trees, we need only one operation, which we call root construction: we give a label f and the nodes $(N_i)_{i<n}$, and we build $\begin{array}{c} f \\ \swarrow \quad \searrow \\ N_0 \cdots N_{n-1} \end{array}$.
Such an operation is constant time in the naive representation and in the sharing representation for finite trees (assuming hashing is constant time [12, 3]). It is indeed also constant time for infinite trees, but this operation does not suffice to build any regular tree. We need also some loop building mechanism. We call this second operation recursive construction. Considering a tree t and a label x, it consists in replacing every edge going to x by an edge to the root, and then apply `representCycle` to maintain the uniqueness of the representation. Concerning the complexity of this algorithm, it seems that the prevailing operation is the final (and unique) call to `share`, which is applied on the smallest possible subgraph, but in the worst case, the quadratic complexity of `shareWithDone` will take precedence.

Many other operations can be adapted to shared trees while preserving the uniqueness of the representation by derivation from the `representation` algorithm. But due to lack of space, we let the reader write their own adaptations.

	sharing representation	naive representation
testing $t_1 = t_2$	$\mathcal{O}(1)$	$\mathcal{O}\left((n_1 + n_2)\log(n_1 + n_2)\right)$
testing t_1 subtree of t_2	$\mathcal{O}(n_2)$	$\mathcal{O}\left((n_1 + n_2)\log(n_1 + n_2)\right)$
building $t_{[p]}$	$\mathcal{O}(\lvert p \rvert)$	$\mathcal{O}(\lvert p \rvert)$
root construction	$\mathcal{O}(1)$	$\mathcal{O}(1)$
recursive construction	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$

Fig. 6. Summary of worst case time complexities

The summary suggests that if we are to perform equality testing, it can be beneficial to perform sharing during the calculus. What we show here are worst case complexity, though, and the difficult cases are quite pathological, and thanks to some simple optimizations, they are quite rare. The situation is very similar to the complexity of operations on BDDs [2] compared to the operations on boolean formulas. The size of the formula representing a given boolean function is unbounded, but the basic operations, like conjunctions, are linear in the size of one of the formulas whereas they are quadratic for the BDDs. Nevertheless, in practice BDDs are far more efficient.

6 Application: Set-Based Analysis

We propose to use these techniques to improve the representations of sets of trees. The expressive power of this improved representation is exactly what is needed in set-based analysis [9], where sets of trees are approximated by ignoring the dependencies between variables (an idea which was already present in [16, 11]).

6.1 Tree Automata and Graphs

Because the cartesian approximation eliminates any dependencies between children of a tree, we can use deterministic top-down tree automata in set-based analysis. The idea we use here is that deterministic top-down tree automata can be seen as graphs, where the only properties that matter are path properties, and so it can be represented efficiently as a regular infinite tree.

A deterministic top-down tree automaton [17, 8] is a tuple (Q, I, δ, F) where Q is a finite set of states, $I \in Q$ is the initial state, $F \subset Q$ is a set of final states, and $\delta : A \times Q \to Q \times \ldots \times Q$ is the transition function which takes a label in A and a state, and returns a sequence of states (as many as the arity of the label). The corresponding graph G is such that $G^N = Q$, $G^E = \{(q, q', a_i) \mid \delta(a, q) = (\ldots, q', \ldots)$ and q' in i^{th} position $\}$. This connection means that we can represent the sets used in set-based analysis without any variable name in the representation, and in a shared way.

6.2 Tree Skeletons

In order to represent the sets of set-based analysis as trees, we use a new label to represent the anonymous states of the tree automata. This label, which we call a choice label corresponds to a possible union in the interpretation of the infinite tree. We denote this label $\bigcirc$. We call the infinite trees with this extra label a tree skeleton. The set of trees represented by a tree skeleton is defined[3] by:

$$\mathrm{Set}\left(\begin{array}{c} f \\ \swarrow\ \searrow \\ t_0 \cdots t_{n-1}\end{array}\right) \stackrel{\text{def}}{=} \left\{ \begin{array}{c} f \\ \swarrow\ \searrow \\ u_0 \cdots u_{n-1}\end{array} \;\middle|\; \forall i < n, u_i \in \mathrm{Set}\,(t_i) \right\}$$

$$\mathrm{Set}\left(\begin{array}{c} \bigcirc \\ \swarrow\ \searrow \\ t_0 \cdots\cdots t_{n-1}\end{array}\right) \stackrel{\text{def}}{=} \bigcup_{i<n} \mathrm{Set}\,(t_i)$$

In order to have a unique representation of the sets of trees (and so keep the constant time equality testing and memoizing properties), we make some restrictions on what infinite trees are considered valid tree skeletons. First we eliminate unnecessary choices: if a choice node has only one child, it is replaced by its child. If a choice node is the child of a choice node, it is replaced by its children. We perform the cartesian approximation: if two children of a choice node have the same label, they are merged (replaced by their cartesian upper approximation). Finally, the children of a choice node are ordered according to their labels. See the summary of figure 7.

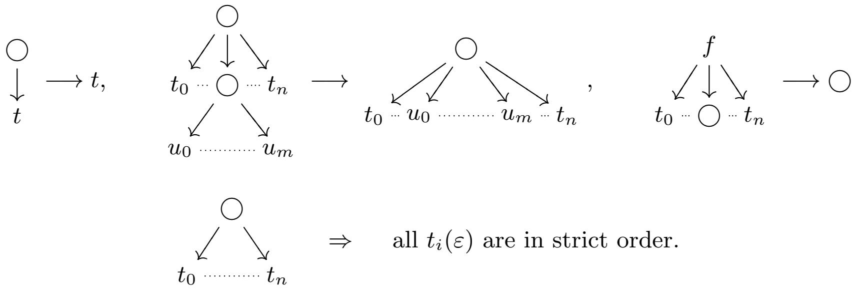


Fig. 7. Rules to obtain a valid tree skeleton

Any deterministic top-down tree automaton can be represented by a valid tree skeleton. Consider an automaton (Q, I, δ, F). We first build the infinite tree labeled by Q and A, such that the root is labeled by I, the children of a given

[3] Set is defined as the least fixpoint of this set of equations. The ordering is the pointwise ordering of the inclusion of the images. If we wanted to include infinite trees (as in [5]), we would take the greatest fixpoint.

state q are the different a such that $\delta(q, a)$ is defined, an the children of such a a are the $\delta(q, a)$. This tree is regular because there is at most one subtree labeled by a given $q \in Q$, and at most $|Q|$ subtrees labeled by a given $a \in A$. The second step consists in removing every label of arity 0 which does not come from a state in F, and in replacing every state by $\bigcirc$. Then we derive the valid tree skeleton.

6.3 Using Tree Skeletons in Analysis

Manipulation of tree skeletons uses basic algorithms on shared infinite regular trees. Once we can keep the maximal sharing property, it is easy to keep track of the other rules for tree skeletons. Then tree skeletons can be used everywhere we consider a set of trees in the analysis. It can replace some of the tree automata of [7] (if we keep the original restrictions of set-based analysis), or the tree grammars of [13], as the approximation on union corresponds indeed to cartesian approximation.

In practice, you can try to use the toolbox under development at the following address: `http://www.di.ens.fr/~mauborgn/skeleton.tar.gz`.

7 Conclusion

While trying to improve the representation of sets of trees in set-based analysis, we presented generic algorithms to manipulate efficiently any structure encoded as infinite regular trees. These algorithms allow a very compact representation of such structures and a constant time equality testing. One of their advantages is their incrementality which allows their use on dynamic structures. The complexity analysis cannot describe the potential benefit of this new representation, but it suggests the same gain as for Binary Decision Diagrams which use similar techniques.

We also described a new way of representing sets of trees using infinite regular trees. This new representation is sharing, incremental and unique. Current work includes the integration of the representation in an actual analyzer to show experimentally its benefits.

Acknowledgments

Many thanks are due to the anonymous referees for their very useful comments.

References

[1] Aho, A. V., Hopcroft, J. E., and Ullman, J. D. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[2] Bryant, R. E. Graph based algorithms for boolean function manipulation. *IEEE Transactions on Computers C-35* (August 1986), 677–691.

[3] Cai, J., and Paige, R. Using multiset discrimination to solve language processing without hashing. *Theoretical Computer Science* (1994). also, U. of Copenhagen Tech. Report, DIKU-TR Num. D-209, 94/16, URL ftp://ftp.diku.dk/diku/semantics/papers/D-209.ps.Z.

[4] Cardon, A., and Crochemore, M. Partitioning a graph in $O(|A| \log_2 |V|)$. *Theoretical Computer Science 19* (1982), 85–98.

[5] Charatonik, W., and Podelski, A. Co-definite set constraints. In *9th International Conference on Rewriting Techniques and Applications* (March-April 1998), T. Nipkow, Ed., vol. 1379 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 211–225.

[6] Colmerauer, A. PROLOG and infinite trees. In *Logic Programming* (1982), K. L. Clark and S.-A. Tärnlund, Eds., vol. 16 of *APIC Studies in Data Processing*, Academic Press, pp. 231–251.

[7] Devienne, P., Talbot, J., and Tison, S. Solving classes of set constraints with tree automata. In *3th International Conference on Principles and Practice of Constraint Programming* (October 1997), G. Smolka, Ed., vol. 1330 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 62–76.

[8] Gécseg, F., and Steinby, M. *Tree Automata.* Akadémia Kiadó, 1984.

[9] Heintze, N. *Set Based Program Analysis.* PhD thesis, School of Computer Science, Carnegie Mellon University, October 1992.

[10] Hopcroft, J. An $n \log n$ algorithm for minimizing states in a finite automaton. In *Theory of machines and computations* (1971), Z. Kohavi and A. Paz, Eds., Academic Press, pp. 189–196.

[11] Jones, N. D., and Muchnick, S. S. Flow analysis and optimization of LISP-like structures. In *6th POPL* (January 1979), ACM Press, pp. 244–256.

[12] Knuth, D. E. *Sorting and Searching*, vol. 3 of *The Art of Computer Programming.* Addison-Wesley, 1973.

[13] Liu, Y. A. Dependence analysis for recursive data. In *IEEE International Conference on Computer Languages* (May 1998), pp. 206–215.

[14] Mauborgne, L. Binary decision graphs. In *Static Analyis Symposium (SAS'99)* (1999), A. Cortesi and G. Filé, Eds., vol. 1694 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 101–116.

[15] Michie, D. "memo" functions and machine learning. *Nature 218* (April 1968), 19–22.

[16] Reynolds, J. Automatic computation of data set definitions. In *Information Processing '68* (1969), Elsevier Science Publisher, pp. 456–461.

[17] Thatcher, J. W., and Wright, J. B. Generalized finite automata with an application to a decision problem of second-order logic. *Mathematical Systems Theory 2* (1968), 57–82.

On the Translation of Procedures to Finite Machines

Abstraction Allows a Clean Proof

Markus Müller-Olm[1] and Andreas Wolf[2] *

[1] Universität Dortmund, Fachbereich Informatik, LS V, 44221 Dortmund, Germany
mmo@ls5.cs.uni-dortmund.de

[2] Christian-Albrechts-Universität, Institut für Informatik und Praktische Mathematik, Olshausenstraße 40, 24098 Kiel, Germany
awo@informatik.uni-kiel.de

Abstract. We prove the correctness of the translation of a prototypic While-language with nested, parameterless procedures to an abstract assembler language with finite stacks. A variant of the well-known wp and wlp predicate transformers, the *weakest relative precondition transformer* wrp, together with a symbolic approach for describing semantics of assembler code allows us to explore assembler programs in a manageable way and to ban finiteness from the scene early.

Keywords: compiler, correctness, refinement, resource-limitation, predicate transformer, procedure, verification

1 Introduction

The construction of compilers is one of the oldest and best studied topics in computer science and neither the interest in this subject nor its importance has declined. Though the range of application of compiler technology has grown, there is still a great need for further understanding the classical setup of program translation. Even if we trust a source program or prove it correct, we cannot rely on the executed object code, if compilation may be erroneous. This motivates us to study the question of how to construct *verified* compilers.

Trusted compilers would permit to certify safety-critical code on the source code level, which promises to be less time-consuming, cheaper, and more reliable than the current practice to inspect the generated machine code [7,13]. The ultimate goal of compiler verification [1,2,4,6,8,9,11,12] is to justify such confidence into compilers.

In [10] we studied the question what semantic relationship we can expect to hold between a target program and the source program from which is was generated. Two natural candidate properties from the point of view of program verification are *preservation of total correctness (PTC)* and *preservation of partial correctness (PPC)*. They require that all total or partial correctness assertions valid for the source program remain valid for the target program. Another

* The work of the second author is supported by DFG grant La 426/15-2.

G. Smolka (Ed.): ESOP/ETAPS 2000, LNCS 1782, pp. 290–304, 2000.

characterization is as refinement of the wp and wlp transformers [3] associated to the source and target program. We argued, however, that neither PTC nor PPC is guaranteed by practical compilers. Limited resources on the target processor prohibit the former: PTC implies that the target program terminates regularly, i.e. without a run-time error, whenever regular termination is guaranteed for the source program. But when we implement a source language with full recursion on a finite machine, "StackOverflow" errors will be observed every now and then. On the other hand, optimizing compilers generally do not preserve partial correctness because common transformations, like dead-code elimination, may eliminate code from the program that causes a run-time error. Thus, run-time errors may be replaced by arbitrary results.

As a remedy we proposed in [10] the more general notion of *preservation of relative correctness (PRC)* (recalled in Sect. 3). Relative correctness is *parameterized* in a set A of *accepted failures* and allows thus – in contrast to partial or total correctness – to treat runtime errors and divergence differently. We also studied a corresponding family of predicate transformers wrp_A. It is convenient to refer to predicate transformer (PT) semantics in compiler proofs because there is a powerful data refinement theory for PTs and refinement proofs can be presented in a calculational style by using algebraic laws [5,6,9]. PTs also interface directly to correctness proofs for source programs. wrp is meant to permit an elegant treatment of runtime errors and finiteness of machines while staying in the familiar and well-studied realm of predicates and predicate transformers.

The main purpose of the current paper is to show that wrp keeps this promise. More specifically, we employ wrp-based reasoning to prove correct the translation of a prototypic While-language with nested, parameterless procedures to an abstract assembler language with finite stacks, a proof that is also of independent interest. We focus on the control flow implementation by jumps and a return address stack. Due to finiteness of stacks, regular termination of target programs generated from terminating source programs cannot be guaranteed. Nevertheless, wrp allows to establish a variant of PTC in which "StackOverflow" is treated as an accepted failure. As intended, finiteness of stacks vanishes from the scene very early: by taking into account that "StackOverflow" is an accepted error, the laws about wrp derived from the operational semantics are akin to the ones of an idealized assembler with unbounded stacks. Thus, wrp allows to reason about implementations on finite machines without burdening the verification. Another interesting aspect of our proof is that we employ *symbolic* ways of reasoning about assembler language semantics instead of referring to more conventional descriptions by means of an instruction pointer.

The remainder of this paper is organized as follows. Sect. 2 recalls the basics of predicates and predicate transformers. Preservation of relative correctness is discussed briefly in Sect. 3. The abstract assembler language which will serve as the target language is presented in Sect. 4 before the source language, a more common high-level language, is introduced in Sect. 5. The translation scheme is defined in Sect. 6 and the actual correctness proof is given in Sections 7 and 8. We conclude with some remarks in Sect. 9.

2 Preliminaries

Predicates. Assume given a set Σ of *states* s; typically a state is a mapping from variables to values. We identify predicates with the set of satisfying states, so *predicates* are of type $\mathit{Pred} = 2^{\Sigma}$ ranged over by ϕ and ψ. *Pred*, ordered by set inclusion, forms a complete Boolean lattice with top-element $\texttt{true} = \Sigma$ and bottom-element $\texttt{false} = \emptyset$.

Predicate transformers. A *predicate transformer (PT)* is a mapping $f : \mathit{Pred} \to \mathit{Pred}$. Sequential composition of two predicate transformers f and g is defined by $(f;g)(\phi) = f(g(\phi))$ and, hence, is associative and has the identity *Id*, $\mathit{Id}(\psi) = \psi$, as unit. We restrict the set of PTs to the monotonic ones because this makes sequential composition monotonic. $\mathit{PTrans} \overset{\text{def}}{=} (\mathit{Pred} \xrightarrow{\text{mon.}} \mathit{Pred})$ together with the lifted order $\leq$ defined by $f \leq g \iff \forall \phi \in \mathit{Pred} : f(\phi) \subseteq g(\phi)$ for $f, g \in \mathit{PTrans}$, is also a complete lattice with top-element $\top$, $\top(\psi) = \texttt{true}$, and bottom-element $\bot$, $\bot(\psi) = \texttt{false}$.

Fixpoints in complete lattices. The famous theorem of Knaster and Tarski ensures that every monotonic function f on a complete lattice $(L, \leq)$ has a least fixpoint μf and a greatest fixpoint νf. A well-known means for proving properties concerning fixpoints is the following.

Theorem 1 (Fixpoint induction). *For $P \subseteq L$ one has $\mu f \in P$ provided that:*

1. $\forall C \subseteq P : C$ *is totally ordered*: $\bigvee C \in P$. *(Admissibility)*
2. $\bot \in P$. *(Base Case)*
3. $\forall x \in P : x \leq f(x) \Longrightarrow f(x) \in P$. *(Induction Step)*

3 Relativized Predicate Transformers

In this section we recall relative correctness and relativized predicate transformers, which were introduced and discussed at length in [10], focusing on what's important for our purposes.

We consider imperative programs π intended to compute on a certain non-empty set of states Σ. For the moment, the details of program execution are not of interest; we are only interested in the final outcomes of computations. We thus assume that each program π is furnished with a relation $R(\pi) \subseteq \Sigma \times (\Sigma \cup \Omega)$, where Ω is a non-empty set of *failure (or irregular) outcomes*[1] disjoint from Σ. Typically, Ω contains error states like "DivByZero" and "StackOverflow" and a special symbol ∞ representing divergence.

We use the following conventions for the naming of variables: Σ is ranged over by s, Ω by ω, and $\Sigma \cup \Omega$ by σ. Intuitively, $(s, s') \in R(\pi)$ records that s' is a possible regular result of π from initial state s, $(s, o) \in R(\pi)$ means that error state $o \in \Omega \setminus \{\infty\}$ can be reached from s, and $(s, \infty) \in R(\pi)$ that π may diverge from s, i.e., run forever. $R(\pi)$ can be thought to be derived from an operational or denotational semantics. An example is discussed in Sect. 4.

[1] We use the more neutral word 'outcome' instead of 'result' because some people object to the idea that divergence is a result of a program.

Relative correctness. When evaluating partial correctness assertions all irregular outcomes of programs are accepted; in contrast in total correctness assertions all irregular outcomes are taken as disproof. Relative correctness is built around the idea of *parameterizing* assertions w.r.t. the set of accepted outcomes. The irregular outcomes that are not accepted are taken as disproof.

Assume given a set $A \subseteq \Omega$ of accepted outcomes; this set may contain divergence as in partial correctness. For a given precondition ϕ and postcondition ψ we call program π *relatively correct w.r.t.* ϕ, ψ *and* A if each π-computation starting in a state satisfying ϕ terminates regularly in a state satisfying ψ or has an accepted outcome in A (e.g. π may diverge if $\infty \in A$). More formally:

$$\langle\phi\rangle\pi\langle\psi\rangle_A \quad \text{iff} \quad \forall s, \sigma : s \in \phi \wedge (s, \sigma) \in R(\pi) \;\Rightarrow\; \sigma \in \psi \cup A \;.$$

The classical notions of partial and total correctness are special cases: partial correctness amounts to $\langle\phi\rangle\pi\langle\psi\rangle_\Omega$ and total correctness to $\langle\phi\rangle\pi\langle\psi\rangle_\emptyset$.

Weakest relative preconditions. Relative correctness gives rise to a corresponding predicate transformer semantics of programs. The *weakest relative precondition* of π w.r.t. ψ and A is the set of regular states from which all π-computations either terminate regularly in a state satisfying ψ or have an outcome in A:

$$\mathsf{wrp}_A(\pi)(\psi) \;=\; \{s \in \Sigma \mid \forall\sigma :\; (s, \sigma) \in R(\pi) \;\Rightarrow\; \sigma \in \psi \cup A\} \;.$$

Note that $\mathsf{wrp}_A(\pi) \in \mathit{PTrans}$. Dijkstra's wlp and wp transformers [3] are just the border cases of wrp: $\mathsf{wrp}_\Omega = \mathsf{wlp}$ and $\mathsf{wrp}_\emptyset = \mathsf{wp}$. There is a fundamental difference between wlp and wp regarding the fixpoint definition of repetitive and recursive construct which generalizes as follows to the wrp_A transformers: if $\infty \in A$ we must refer to greatest fixpoints, otherwise to least ones.

The following equivalence generalizes the well-known characterization of partial and total correctness in terms of wlp and wp:

$$\phi \subseteq \mathsf{wrp}_A(\pi)(\psi) \qquad \Longleftrightarrow \qquad \langle\phi\rangle\pi\langle\psi\rangle_A \;.$$

Preserving relative correctness. A natural way to approach translation correctness is to focus on *properties* that transfer from source to target programs. Suppose, for instance, that π is a source program and π' is its translation. We say that the translation *preserves relative correctness w.r.t.* A if

$$\forall\phi, \psi :\; \langle\phi\rangle\pi\langle\psi\rangle_A \;\Rightarrow\; \langle\phi\rangle\pi'\langle\psi\rangle_A \;, \tag{1}$$

i.e., if all relative correctness assertions transfer from π to π'. It is straightforward to show that (1) is equivalent to the refinement inequality $\mathsf{wrp}_A(\pi) \leq \mathsf{wrp}_A(\pi')$. Refinement between predicate transformers can be established by algebraic calculations. We can thus take advantage from such algebraic calculations in semantic compiler proofs. The remaining part of this section is devoted to providing suitable notations that enable this in the scenario studied in this paper.

Concrete predicate transformers. Suppose given three basic sets of syntactic objects: a set *Var* of *variables* x, a set *Expr* of *expressions* e, and a set *BExpr* of *Boolean expressions* b. We assume interpretation functions for expressions and Boolean expressions $\mathcal{E}(e) : \Sigma \to (\mathit{Val} \cup \Omega)$ and $\mathcal{B}(b) : \Sigma \to (\mathbb{B} \cup \Omega)$; here *Val* is the value set of variables and the set $\mathbb{B} = \{\mathsf{tt}, \mathsf{ff}\}$ represents the truth values. For the remainder of this paper, states are valuations of variables, i.e. $\Sigma = (\mathit{Var} \to \mathit{Val})$. Intuitively, results $\mathcal{E}(e)(s), \mathcal{B}(b)(s) \in \Omega$ represent failures (including divergence) arising during evaluation of (Boolean) expressions.

In order to deal with partially defined expressions we assume special types of basic predicates: $\mathsf{def}(e) \stackrel{\text{def}}{=} \{s \mid \mathcal{E}(e)(s) \in \mathit{Val}\}$ and $\mathsf{in}_A(e) \stackrel{\text{def}}{=} \{s \mid \mathcal{E}(e)(s) \in A\}$ for expressions e and $A \subseteq \Omega$. Analogously, we have the predicates $\mathsf{def}(b)$, $\mathsf{in}_A(b)$ and also $b = \mathsf{tt} \stackrel{\text{def}}{=} \{s \mid \mathcal{B}(b)(s) = \mathsf{tt}\}$ and $b = \mathsf{ff}$ for Boolean expressions b.

We consider an assignment $x := e$. The expression e is evaluated in some given state, and if evaluation delivers a regular result it is assigned to x. But evaluation of e might also fail with an outcome ω. It depends on whether $\omega \in A$ or not if we consider this acceptable. Hence, the weakest relative precondition of this assignment w.r.t. A and postcondition ψ is given by

$$(x :=_A e)(\psi) \stackrel{\text{def}}{=} \mathsf{in}_A(e) \cup (\mathsf{def}(e) \cap \psi[e/x]) \ .$$

Another example is a conditional with branches P and Q guarded by b, where the PTs P and Q are wrp_A-transformers. Obviously the weakest relative precondition w.r.t. A and ψ of this construct is $P(\psi)$ resp. $Q(\psi)$ if b evaluates to tt resp. ff. Since evaluation of b can also fail, the weakest relative precondition PT w.r.t. A and postcondition ψ is given by

$$(P \lhd b/A \rhd Q)(\psi) \stackrel{\text{def}}{=} \mathsf{in}_A(b) \cup (b = \mathsf{tt} \cap P(\psi)) \cup (b = \mathsf{ff} \cap Q(\psi)) \ .$$

4 An Abstract Assembler Language

Syntax. The language defined in this section is intended to capture the essence of flat, unstructured assembler code. In this, our main interest is a realistic treatment of control structures. Therefore, labels l taken from a set *Lab* are used to mark the destination of jump instructions as common in assembler languages. In order to keeps things manageable, the language works on a state space with named variables and we provide instructions embodying entire (Boolean) expressions: $\mathsf{asg}(x, e)$ and $\mathsf{cj}(b, l)$. Such instructions should be thought to be 'macros' representing a sequence of more concrete assembler instructions. A language of this kind might be used as a stepping stone on the way down to actual binary machine code.

The set *Instr* consists of *instructions* of the following form.

- $\mathsf{asg}(x, e)$: an assignment instruction,
- $\mathsf{cj}(b, l)$: a conditional jump (on false) to label l,
- $\mathsf{jsr}(l)$: a subroutine jump to label l, and
- ret: a return jump.

We write $\mathsf{goto}(l)$ for $\mathsf{cj}(\texttt{false}, l)$. It represents an unconditional jump.

An *assembler (or machine) program* m is a finite sequence consisting of instructions and labels where we assume *unique labeling*, formally

$$m \in MP \overset{\text{def}}{=} \{m \in (Instr \cup Lab)^* \mid \forall i, j : m_i = m_j \in Lab \ \Rightarrow \ i = j\} \ .$$

Concatenation of programs is denoted by an infix dot "·". A program m is called *closed* if every label that has an applied occurrence in m also has a defining occurrence. The set of closed programs is denoted by *CMP*. Here is an example of a closed program computing the factorial of x leaving the result in variable y:

$$\mathsf{asg}(\mathsf{y}, 1) \cdot \mathsf{Loop} \cdot \mathsf{cj}(\mathsf{x} \neq 0, \mathsf{End}) \cdot \mathsf{asg}(\mathsf{y}, \mathsf{x} * \mathsf{y}) \cdot \mathsf{asg}(\mathsf{x}, \mathsf{x} - 1) \cdot \mathsf{goto}(\mathsf{Loop}) \cdot \mathsf{End}$$

Basic operational semantics. A processor executing a machine program will typically use an instruction pointer that points to the next instruction to be executed at any given moment. For reasoning about assembler code, however, it is more convenient to represent the current control point in a more symbolic manner: we partition the executed program m into two parts u, v such that $m = u \cdot v$ and that the next instruction to be executed is just the first instruction of v. Progress of execution can nicely be expressed by partitioning the same code sequence differently. *PMP* (partitioned machine programs) denotes the set of pairs $\langle u, v\rangle$ such that $u \cdot v \in CMP$.

Similarly, we prefer to work with a symbolic representation of the stack of return addresses; such a stack is necessary to execute jump-subroutine and return instructions. The idea is to use a stack of partitioned code sequences (modeled by a member of PMP^*) instead of a stack of addresses.

The basic semantics of the abstract assembler language is an operational semantics built around the ideas just described. It works on configurations of the form $\langle u, v, a, s\rangle$, where $\langle u, v\rangle \in PMP$ models the current control point ($u \cdot v$ is the executed program), $a \in PMP^*$ is the symbolic representation of the return stack, and $s \in \Sigma$ is the current state. Thus,

$$\Gamma \overset{\text{def}}{=} \{\langle u, v, a, s\rangle \mid \langle u, v\rangle \in PMP \wedge a \in PMP^* \wedge s \in \Sigma\}$$

is the set of *regular configurations*. In order to treat error situations, we use the members of Ω as *irregular configurations*. Table 1 defines the transition relation $\rightarrow \subseteq \Gamma \times (\Gamma \cup \Omega)$ of an abstract machine executing assembler programs.

Let us consider the rules in more detail. [Asg1] applies if expression e evaluates without error to a value in the current state s: the machine changes the value of x accordingly – the new state is $s[x \mapsto \mathcal{E}(e)(s)]$ – and transfers control to the subsequent instruction by moving $\mathsf{asg}(x, e)$ to the end of the u-component. [Asg2] is used if evaluation of e fails in the current state: the failure value $\mathcal{E}(e)(s)$ is just propagated. [Cj1] describes that a conditional jump $\mathsf{cj}(b, l)$ is not taken if b evaluates to tt in the current state: control is simply transferred to the subsequent instruction. If b evaluates to ff, rule [Cj2] applies and the control is transferred to label l, the position of which is determined by the premise $u \cdot \mathsf{cj}(b, l) \cdot v = x \cdot l \cdot y$. [CJ3] propagates errors resulting from evaluation of b. [Jsr1] is concerned with a

Table 1. Operational semantics of the assembler language

$$[\text{Asg1}]\quad \frac{\mathcal{E}(e)(s) \in \Sigma}{\langle u, \mathsf{asg}(x,e)\cdot v, a, s\rangle \rightarrow \langle u\cdot \mathsf{asg}(x,e), v, a, s[x \mapsto \mathcal{E}(e)(s)]\rangle}$$

$$[\text{Asg2}]\quad \frac{\mathcal{E}(e)(s) \in \Omega}{\langle u, \mathsf{asg}(x,e)\cdot v, a, s\rangle \rightarrow \mathcal{E}(e)(s)}$$

$$[\text{Cj1}]\quad \frac{\mathcal{B}(b)(s) = \mathsf{tt}}{\langle u, \mathsf{cj}(b,l)\cdot v, a, s\rangle \rightarrow \langle u\cdot \mathsf{cj}(b,l), v, a, s\rangle}$$

$$[\text{Cj2}]\quad \frac{\mathcal{B}(b)(s) = \mathsf{ff}\ ,\ u\cdot \mathsf{cj}(b,l)\cdot v = x\cdot l\cdot y}{\langle u, \mathsf{cj}(b,l)\cdot v, a, s\rangle \rightarrow \langle x, l\cdot y, a, s\rangle}$$

$$[\text{Cj3}]\quad \frac{\mathcal{B}(b)(s) \in \Omega}{\langle u, \mathsf{cj}(b,l)\cdot v, a, s\rangle \rightarrow \mathcal{B}(b)(s)}$$

$$[\text{Jsr1}]\quad \frac{u\cdot \mathsf{jsr}(l)\cdot v = x\cdot l\cdot y}{\langle u, \mathsf{jsr}(l)\cdot v, a, s\rangle \rightarrow \langle x, l\cdot y, a\cdot\langle u\cdot \mathsf{jsr}(l), v\rangle, s\rangle}$$

$$[\text{Jsr2}]\quad \langle u, \mathsf{jsr}(l)\cdot v, a, s\rangle \rightarrow \text{``}\mathsf{StackOverflow}\text{''}$$

$$[\text{Ret1}]\quad \langle u, \mathsf{ret}\cdot v, a\cdot\langle x, y\rangle, s\rangle \rightarrow \langle x, y, a, s\rangle$$

$$[\text{Ret2}]\quad \langle u, \mathsf{ret}\cdot v, \varepsilon, s\rangle \rightarrow \text{``}\mathsf{EmptyStack}\text{''}$$

$$[\text{Label}]\quad \langle u, l\cdot v, a, s\rangle \rightarrow \langle u\cdot l, v, a, s\rangle$$

subroutine jump to label l. Similarly to rule [Cj2], control is transferred to label l. Additionally, the machine stores the return address by pushing $\langle u\cdot \mathsf{jsr}(l), v\rangle$ onto the symbolically modeled return stack a. If execution subsequently reaches a ret instruction, execution of $\langle u\cdot \mathsf{jsr}(l), v\rangle$ is resumed as specified by [Ret1]. A processor with finite memory will not always be able to stack a return address when executing a jsr instruction. We model this by rule [Jsr2] that allows the machine to report "StackOverflow" spontaneously. Of course, in an actual processor the choice between regular stacking and overflow will be mutually exclusive and not just non-deterministic as in our model. This could be modeled by furnishing [Jsr2] by a premise *StackFull* and [Jsr1] by a premise $\neg$*StackFull*, where *StackFull* is a (complicated) condition depending on the current state of the machine. Finally, [Ret2] reports an error if a ret instruction is executed on an empty return stack, and [Label] allows to skip labels.

The evaluation of m in state s starts in the initial configuration $\langle \varepsilon, m, \varepsilon, s\rangle$, i.e. with the first instruction of m and with an empty stack. Execution terminates regularly if a configuration of the form $\langle u, \varepsilon, a, s'\rangle$ is reached; other possible outcomes are reachable error configurations ω, and ∞, if there is an infinite sequence of transitions from $\langle \varepsilon, m, \varepsilon, s\rangle$. Based on this intuition, we could now define a relational semantics $R(m)$ for a given program $m \in \mathit{CMP}$ following the lines of the definition below. $R(m)$ would give rise to a family of predicate transformers $\mathsf{wrp}_A(m)$. Up to this point $\mathsf{wrp}_A(m)$ would be known only with reference to the operational semantics. In order to allow a reasoning on a more abstract level we would like to derive sufficiently strong laws about $\mathsf{wrp}_A(m)$ from the operational semantics first; afterwards we would use just these laws in our reasoning without referring directly to the operational semantics.

Unfortunately, this approach fails for $\mathsf{wrp}_A(m)$: only very weak laws can be established. The main problem is that the behavior of jump and jump-subroutine instructions cannot adequately be described without having context information available. We, therefore, work with a semantics of machine programs that takes the sequential context as well as the stack context into account.

For $\langle u, v\rangle \in PMP$ and $a \in PMP^*$ we define

$$\begin{aligned} R(u,v,a) \overset{\text{def}}{=} \quad & \{(s,s') \mid \exists u', a' : \langle u,v,a,s\rangle \rightarrow^* \langle u', \varepsilon, a', s'\rangle\} \\ & \cup \{(s,\omega) \mid \langle u,v,a,s\rangle \rightarrow^* \omega\} \\ & \cup \{(s,\infty) \mid \langle u,v,a,s\rangle \rightarrow^\infty\} \ , \end{aligned}$$

where $\rightarrow^*$ denotes the reflexive and transitive closure of $\rightarrow$, and $\rightarrow^\infty$ the existence of an infinite path. This definition induces a family of predicate transformers $\mathsf{wrp}_A(u,v,a)$ and it is this family that we are using in our reasoning. We can define $R(m)$ and $\mathsf{wrp}_A(m)$ by $R(m) = R(\varepsilon, m, \varepsilon)$ and $\mathsf{wrp}_A(m) = \mathsf{wrp}_A(\varepsilon, m, \varepsilon)$.

The laws in Table 2 can now be proved from the operational semantics. Technically these laws are just derived properties but they can also be read as axioms about the total behavior of a machine. Law [Asg-wrp], e.g., tells us about a machine started in a situation where it executes an asg-instruction first: its total behavior can safely be assumed to be composed of the respective assignment to x and the total behavior of a machine started just after the assignment instruction. The other laws have a similar interpretation. Together the laws allow a kind of symbolic execution of assembler programs. But we do not have to refer to low-level concepts like execution sequences; instead we can use more abstract properties, e.g., that $\geq$ is an ordering.

All these laws can be strengthened to equalities. We state them as inequalities in order to stress that just one direction is needed in the following. Refinement allows to use safe approximations on the right hand side instead of fully accurate descriptions. This allows to reason safely about instructions whose effect is either difficult to capture or not fully specified by the manufacturer of the processor [9]. If, for example, [Jsr1] and [Jsr2] are furnished with a condition *StackFull* as discussed above, the refinement inequality stated in [Jsr-wrp] becomes proper, because jsr would definitely lead to the acceptable error "StackOverflow" if *StackFull* holds. Therefore, the PT on the left hand side would succeed for all states satisfying *StackFull* irrespective of the post-condition, while the right hand side may fail.

Note that the premise "StackOverflow" $\in A$ of the law [Jsr-wrp] is essential. If "StackOverflow" is considered unacceptable ("StackOverflow" $\notin A$), we have $\mathsf{wrp}_A(u, \mathsf{jsr}(l) \cdot v, a) = \bot$ as a consequence of [Jsr2]. This means that jsr cannot be used to implement any non-trivial statement. If the more precise operational model with a *StackFull* predicate is used, $\mathsf{wrp}_A(u, \mathsf{jsr}(l) \cdot v, a)$ is better than $\bot$ but any non-trivial approximation will involve the *StackFull* predicate. This would force us to keep track of the storage requirements when we head for a verified

Table 2. wrp-laws for the assembler language

[Asg-wrp]	$\mathsf{wrp}_A(u, \mathsf{asg}(x, e) \cdot v, a)$	$\geq (x :=_A e) \; ; \mathsf{wrp}_A(u \cdot \mathsf{asg}(x, e), v, a)$
[Cj-wrp]	$\mathsf{wrp}_A(u, \mathsf{cj}(b, l) \cdot v, a)$	$\geq \mathsf{wrp}_A(u \cdot \mathsf{cj}(b, l), v, a) \lhd b/A \rhd \mathsf{wrp}_A(x, l \cdot y, a)$, if $u \cdot \mathsf{cj}(b, l) \cdot v = x \cdot l \cdot y$
[Goto-wrp]	$\mathsf{wrp}_A(u, \mathsf{goto}(l) \cdot v, a)$	$\geq \mathsf{wrp}_A(x, l \cdot y, a)$, if $u \cdot \mathsf{goto}(l) \cdot v = x \cdot l \cdot y$
[Jsr-wrp]	$\mathsf{wrp}_A(u, \mathsf{jsr}(l) \cdot v, a)$	$\geq \mathsf{wrp}_A(x, l \cdot y, a \cdot \langle u \cdot \mathsf{jsr}(l), v \rangle)$, if $u \cdot \mathsf{jsr}(l) \cdot v = x \cdot l \cdot y$ and "StackOverflow" $\in A$
[Ret-wrp]	$\mathsf{wrp}_A(u, \mathsf{ret} \cdot v, a \cdot \langle x, y \rangle)$	$\geq \mathsf{wrp}_A(x, y, a)$
[Label-wrp]	$\mathsf{wrp}_A(u, l \cdot v, a)$	$\geq \mathsf{wrp}_A(u \cdot l, v, a)$
[Term-wrp]	$\mathsf{wrp}_A(u, \varepsilon, a)$	$\geq \mathit{Id}$

compilation. As the recursion depth of programs is in general not computable, we could not justify translation of arbitrary recursive procedures.

5 A Simple High-Level Language

As a prototypic instance of a high-level language we consider a While-language with parameterless, nested procedures. Such a language is adequate for studying the control-flow aspects of translation of ALGOL-like programming languages.

Syntax. We define the set of programs, *Prog*, by the following grammar. In order to distinguish programs clearly from the corresponding semantic predicate transformers from Sect. 3 we use an abstract kind of syntax.

$$\pi ::= \mathsf{assign}(x, e) \mid \mathsf{seq}(\pi_1, \pi_2) \mid \mathsf{if}(b, \pi_1, \pi_2) \mid \mathsf{while}(b, \pi) \mid \mathsf{call}(p) \mid \mathsf{blk}(p, \pi_p, \pi_b)$$

In this grammar, x ranges over the variables in *Var*, b and e over *BExpr* and *Expr*, and p over a set *ProcName* of procedure identifiers.

$\mathsf{blk}(p, \pi_p, \pi_b)$ is a block in which a (possibly recursive) local procedure p with body π_p is declared. π_b is the body of the block; it might call p as well as more globally defined procedures. The semantics below ensures static scoping and so the translation of the next section has to guarantee static scoping as well. Note that nesting of procedure declarations and even re-declaration is allowed. Our exposition generalizes straightforwardly to blocks in which a system of mutually recursive procedures can be declared instead of just a single procedure. We refrained from treating this more general case only, as it burdens the notation a bit without bringing more insight. The intuitive semantics of the other syntactic operators should be clear from their name.

Semantics. Now we furnish the While-language with a predicate transformer semantics. Due to lack of space, we cannot follow the lines from the last section;

instead we *postulate* the resulting predicate transformer semantics directly. Nevertheless the oncoming definitions should be read as *laws* derived from a more concrete semantics. In [10] we justified such definitions briefly for a language without procedures.

In order to give a compositional semantics, we refer as usual to *environments* $\eta \in \mathit{Env} \overset{\text{def}}{=} (\mathit{ProcName} \to \mathit{PTrans})$, mapping procedure identifiers to the weakest relative precondition transformer of their body. The environment is taken by wrp as an additional argument written as a superscript.

$$\begin{aligned}
\mathsf{wrp}^{\eta}_{A}(\mathsf{assign}(x,e)) &= (x :=_A e)\\
\mathsf{wrp}^{\eta}_{A}(\mathsf{seq}(\pi_1,\pi_2)) &= \mathsf{wrp}^{\eta}_{A}(\pi_1)\ ;\ \mathsf{wrp}^{\eta}_{A}(\pi_2)\\
\mathsf{wrp}^{\eta}_{A}(\mathsf{if}(b,\pi_1,\pi_2)) &= \mathsf{wrp}^{\eta}_{A}(\pi_1) \lhd b/A \rhd \mathsf{wrp}^{\eta}_{A}(\pi_2)\\
\mathsf{wrp}^{\eta}_{A}(\mathsf{while}(b,\pi)) &= \lambda\mathcal{W}\\
\mathsf{wrp}^{\eta}_{A}(\mathsf{call}(p)) &= \eta(p)\\
\mathsf{wrp}^{\eta}_{A}(\mathsf{blk}(p,\pi_p,\pi_b)) &= \mathsf{wrp}^{\eta[p\mapsto\lambda\mathcal{P}]}_{A}(\pi_b)
\end{aligned}$$

In the clauses for while and blk, $\lambda = \nu$ if $\infty \in A$, and $\lambda = \mu$ otherwise, i.e. we have to take the greatest fixpoint if divergence is accepted (like in partial correctness semantics) and the least fixpoint otherwise (see [10]).

Let us discuss briefly each of the clauses in turn. The assignment law takes advantage from the assignment combinator defined in Sect. 3. The weakest precondition of a sequential composition is the weakest precondition of the first statement establishing the weakest precondition of the second statement. A conditional's weakest precondition depends on the validity of the guard. Operationally a loop is unrolled as long as the guard holds, hence the weakest precondition PT of a loop is a fixpoint of the well known semantical function $\mathcal{W} : \mathit{PTrans} \to \mathit{PTrans}$, where $\mathcal{W}(X) = (\pi; X) \lhd b/A \rhd \mathit{Id}$. Application of the environment in question captures the call-case. A block's weakest precondition in some given environment is the weakest precondition of the body in a varied environment that contains a new binding for the local procedure declared in that block. The weakest precondition of that procedure is a fixpoint of the function $\mathcal{P} : \mathit{PTrans} \to \mathit{PTrans}$, where $\mathcal{P}(X) = \mathsf{wrp}^{\eta[p\mapsto X]}_{A}(\pi_p)$.

Complete programs are interpreted in the environment $\bot_{\mathit{Env}}$ that bind all procedures to the $\bot$ predicate transformer, because otherwise the call of an undeclared procedure would miraculously have a non-trivial meaning. Hence, when comparing a complete program π to its translation, we refer to $\mathsf{wrp}^{\bot_{\mathit{Env}}}_{A}(\pi)$.

6 Specification of Compilation

In Table 3 we inductively define a compiling relation $\mathcal{C} \subseteq \mathit{Prog} \times \mathit{MP} \times \mathit{Dict}$. Here $\mathit{Dict} = (\mathit{ProcName} \overset{\text{fin}}{\to} \mathit{Lab})$ is the set of *dictionaries* that intuitively map procedure names to labels where code for the corresponding body can be found. We have $\mathcal{C}(\pi, m, \delta)$ if machine program m is a possible compiling result of

Table 3. Compiling relation

$$\text{[Assign]}\quad \mathcal{C}(\mathsf{assign}(x,e),\mathsf{asg}(x,e),\delta)$$

$$\text{[Seq]}\quad \frac{\mathcal{C}(\pi_1,m_1,\delta),\ \mathcal{C}(\pi_2,m_2,\delta)}{\mathcal{C}(\mathsf{seq}(\pi_1,\pi_2),m_1\cdot m_2,\delta)}$$

$$\text{[If]}\quad \frac{\mathcal{C}(\pi_1,m_1,\delta),\ \mathcal{C}(\pi_2,m_2,\delta)}{\mathcal{C}(\mathsf{if}(b,\pi_1,\pi_2),\mathsf{cj}(b,l_1)\cdot m_1\cdot \mathsf{goto}(l_2)\cdot l_1\cdot m_2\cdot l_2,\delta)}$$

$$\text{[While]}\quad \frac{\mathcal{C}(\pi,m,\delta)}{\mathcal{C}(\mathsf{while}(b,\pi),l_0\cdot \mathsf{cj}(b,l_1)\cdot m\cdot \mathsf{goto}(l_0)\cdot l_1,\delta)}$$

$$\text{[Call]}\quad \frac{p\in \mathrm{dom}(\delta)}{\mathcal{C}(\mathsf{call}(p),\mathsf{jsr}(\delta(p)),\delta)}$$

$$\text{[Blk]}\quad \frac{\mathcal{C}(\pi_p,m_p,\delta[p\mapsto l_p]),\ \mathcal{C}(\pi_b,m_b,\delta[p\mapsto l_p])}{\mathcal{C}(\mathsf{blk}(p,\pi_p,\pi_b),\mathsf{goto}(l_b)\cdot l_p\cdot m_p\cdot \mathsf{ret}\cdot l_b\cdot m_b,\delta)}$$

source program π assuming that dictionary δ assigns appropriate labels to free procedure names. The program

$$\mathsf{seq}(\mathsf{assign}(\mathsf{y},1),\mathsf{while}(\mathsf{x}>0,\mathsf{seq}(\mathsf{assign}(\mathsf{y},\mathsf{x}*\mathsf{y}),\mathsf{assign}(\mathsf{x},\mathsf{x}-1))))\ ,$$

for instance, may be compiled to the assembler program computing the factorial function in Sect. 4 irrespective of the dictionary δ.

Note that the typing constraint $m \in \mathit{MP}$ guarantees that target programs are labeled uniquely. An advantage of a relational specification over a functional compiling-function is that certain aspects, like choice of labels here, can be left open for a later design stage of the compiler.

7 Correctness of Compilation

This section is concerned with proving correctness of the translation specified in the previous section. As discussed in the introduction, the translation cannot be correct in the sense of preservation of total correctness (PTC), as our assembler language might report "StackOverflow" on executing a jsr instruction and thus regularly terminating source programs might be compiled to target programs that do not terminate regularly. Nevertheless source programs that do not diverge are never compiled to diverging target programs. But PTC identifies divergence and runtime-errors and, therefore, it cannot treat this scenario appropriately. A main purpose of this paper is to show how the greater selectivity of wrp_A-based reasoning allows a more adequate treatment by appropriate choice of A. We treat "StackOverflow" as an acceptable outcome but ∞ as an unacceptable one. This gives rise to a relativized version of PTC. We comment on the proof for relativized versions of PPC in the conclusion.

Theorem 2. *Suppose* $\infty \notin A$ *and* "$\mathsf{StackOverflow}$" $\in A$. *Then for all* π, m:

$$\mathcal{C}(\pi, m, \emptyset) \;\Rightarrow\; \mathsf{wrp}_A(m) \geq \mathsf{wrp}_A^{\perp_{Env}}(\pi) \ .$$

Thus, if a program π is compiled to an assembler program m in an empty dictionary, relative correctness is preserved. Note that the premise of the compiling rule [Call] guarantees, that non-closed programs cannot be compiled with an empty dictionary.

When we try to prove Theorem 2 by a structural induction we encounter two problems. Firstly, when machine programs are put together to implement composed programs, like in the [Seq] or [If] rule, the induction hypothesis cannot directly be applied because it is concerned with code for the components in isolation while, in the composed code, the code runs in the context of other code. Our approach to deal with this problem is to establish a stronger claim that involves a universal quantification over all contexts. More specifically, we show $\mathsf{wrp}_A(u, m \cdot v, a) \geq \mathsf{wrp}_A^{\eta}(\pi)\ ;\ \mathsf{wrp}_A(u \cdot m, v, a)$ for all surrounding code sequences u, v and stack contexts a. Note how the sequential composition with $\mathsf{wrp}_A(u \cdot m, v, a)$ on the right hand side beautifully expresses that m transfers control to the subsequent code and that the stack is left unchanged.

Secondly, when considering the call-case, some knowledge about the bindings in the dictionary δ is needed. To solve this problem we use the following predicate.

$$\begin{aligned}\mathsf{fit}(\eta, \delta, u) \;\overset{\text{def}}{\Longleftrightarrow}\; & \forall q \in \mathrm{dom}(\delta) : \exists x, y : \\ & \quad x \cdot \delta(q) \cdot y = u \;\;\wedge \\ & \quad \forall e, f, g :\; \mathsf{wrp}_A(x, \delta(q) \cdot y, g \cdot \langle e, f \rangle) \geq \eta(q)\ ;\ \mathsf{wrp}_A(e, f, g) \ .\end{aligned}$$

It expresses that the bindings in δ together with the assembler code u 'fit' to the bindings in the semantic environment η. The first conjunct says that the context provides a corresponding label for each procedure q bound by δ; the second conjunct tells us that the code following this label implements q's binding in η and proceeds with the code on top of the return stack. This is just what is needed in the call-case of the induction. The code generated for blocks has to ensure that this property remains valid for newly declared procedures.

Putting the pieces together we are going to prove the following.

Lemma 3. *Suppose* $\infty \notin A$ *and* "$\mathsf{StackOverflow}$" $\in A$. *For all* $\pi, m, u, v, a, \eta, \delta$:

$$\mathcal{C}(\pi, m, \delta) \wedge \mathsf{fit}(\eta, \delta, u \cdot m \cdot v) \;\Rightarrow\; \mathsf{wrp}_A(u, m \cdot v, a) \geq \mathsf{wrp}_A^{\eta}(\pi)\ ;\ \mathsf{wrp}_A(u \cdot m, v, a) \ .$$

Theorem 2 follows by the instantiation $u = v = \varepsilon$, $a = \varepsilon$, $\eta = \perp_{Env}$, $\delta = \emptyset$ using the [Term-wrp] law and the fact that $\mathsf{wrp}_A(m) = \mathsf{wrp}_A(\varepsilon, m, \varepsilon)$.

8 Proof of Lemma 3

The proof is by structural induction on π. So consider some arbitrarily chosen $\pi, m, u, v, a, \eta, \delta$ such that $\mathcal{C}(\pi, m, \delta)$ and $\mathsf{fit}(\eta, \delta, u \cdot m \cdot v)$, and assume that for all component programs the claim of Lemma 3 holds. As usual, we proceed by a case analysis on the structure of π. In each case we perform a kind of 'symbolic execution' of the corresponding assembler code using the wrp-laws from Sect. 4. The assumptions about fit will solve the call-case elegantly, the while- and blk-case moreover involve some fixpoint reasoning.

Due to lack of space we can discuss here only the cases concerned with procedures: call and blk.

Case a.) $\pi = \mathsf{call}(p)$. By the [Call] rule, $m = \mathsf{jsr}(\delta(p))$ and $p \in \mathrm{dom}(\delta)$. As a consequence of $\mathsf{fit}(\eta, \delta, u \cdot m \cdot v)$ there exist x and y such that $x \cdot \delta(p) \cdot y = u \cdot \mathsf{jsr}(\delta(p)) \cdot v$. Now,

$$\begin{aligned}
& \mathsf{wrp}_A(u, \mathsf{jsr}(\delta(p)) \cdot v, a) \\
\geq \quad & \{\text{Law [Jsr-wrp], ``}\mathsf{StackOverflow}\text{''} \in A\text{, existence of } x \text{ and } y\} \\
& \mathsf{wrp}_A(x, \delta(p) \cdot y, a \cdot \langle u \cdot \mathsf{jsr}(\delta(p)), v\rangle) \\
\geq \quad & \{\text{Second conjunct of } \mathsf{fit}(\eta, \delta, u \cdot m \cdot v)\} \\
& \eta(p) \,;\, \mathsf{wrp}_A(u \cdot \mathsf{jsr}(\delta(p)), v, a) \\
= \quad & \{\text{Definition of call semantics}\} \\
& \mathsf{wrp}^{\eta}_A(\pi) \,;\, \mathsf{wrp}_A(u \cdot \mathsf{jsr}(\delta(p)), v, a) \ .
\end{aligned}$$

Case b.) $\pi = \mathsf{blk}(p, \pi_p, \pi_b)$. By the [Blk] rule, there are assembler programs m_p, m_b and labels l_p, l_b such that $m = \mathsf{goto}(l_b) \cdot l_p \cdot m_p \cdot \mathsf{ret} \cdot l_b \cdot m_b$ and $\mathcal{C}(\pi_p, m_p, \delta[p \mapsto l_p])$ and $C(\pi_b, m_b, \delta[p \mapsto l_p])$ hold.

We would like to calculate as follows:

$$\begin{aligned}
& \mathsf{wrp}_A(u, \mathsf{goto}(l_b) \cdot l_p \cdot m_p \cdot \mathsf{ret} \cdot l_b \cdot m_b \cdot v, a) \\
\geq \quad & \{\text{Laws [Goto-wrp] and [Label-wrp]}\} \\
& \mathsf{wrp}_A(u \cdot \mathsf{goto}(l_b) \cdot l_p \cdot m_p \cdot \mathsf{ret} \cdot l_b, m_b \cdot v, a) \\
\geq \quad & \{\text{Induction hypothesis: } \mathcal{C}(\pi_b, m_b, \delta[p \mapsto l_p]) \text{ holds}\} \\
& \mathsf{wrp}^{\eta[p \mapsto \mu\mathcal{P}]}_A(\pi_b) \,;\, \mathsf{wrp}_A(u \cdot \mathsf{goto}(l_b) \cdot l_p \cdot m_p \cdot \mathsf{ret} \cdot l_b \cdot m_b, v, a) \\
= \quad & \{\text{Definition of block semantics}\} \\
& \mathsf{wrp}^{\eta}_A(\mathsf{blk}(p, \pi_p, \pi_b)) \,;\, \mathsf{wrp}_A(u \cdot \mathsf{goto}(l_b) \cdot l_p \cdot m_p \cdot \mathsf{ret} \cdot l_b \cdot m_b, v, a) \ .
\end{aligned}$$

In order to apply the induction hypothesis in the second step, however, we have to check $\mathsf{fit}(\eta[p \mapsto \mu\mathcal{P}], \delta[p \mapsto l_p], u \cdot m \cdot v)$, i.e. that for all $q \in \mathrm{dom}(\delta[p \mapsto l_p])$

$$\begin{aligned}
& \exists x, y : \\
& \quad x \cdot \delta[p \mapsto l_p](q) \cdot y = u \cdot m \cdot v \ \wedge \\
& \quad \forall e, f, g : \ \mathsf{wrp}_A(x, \delta[p \mapsto l_p](q) \cdot y, g \cdot \langle e, f\rangle) \ \geq \ \eta[p \mapsto \mu\mathcal{P}](q) \,;\, \mathsf{wrp}_A(e, f, g) \ .
\end{aligned} \tag{2}$$

So suppose given $q \in \mathrm{dom}(\delta[p \mapsto l_p])$. If $q \neq p$, (2) reduces to

$$\begin{aligned}
\exists x, y : \ & x \cdot \delta(q) \cdot y = u \cdot m \cdot v \ \wedge \\
& \quad \forall e, f, g : \ \mathsf{wrp}_A(x, \delta(q) \cdot y, g \cdot \langle e, f\rangle) \ \geq \ \eta(q) \,;\, \mathsf{wrp}_A(e, f, g) \ ,
\end{aligned}$$

which follows directly from $\mathsf{fit}(\eta, \delta, u \cdot m \cdot v)$. For $q = p$, on the other hand, we must prove

$$\begin{aligned}\exists x, y :\ & x \cdot l_p \cdot y = u \cdot m \cdot v \ \wedge \\ & \forall e, f, g :\ \mathsf{wrp}_A(x, l_p \cdot y, g \cdot \langle e, f\rangle) \geq \mu\mathcal{P} \ ; \mathsf{wrp}_A(e, f, g) \ .\end{aligned}$$

Choosing $x = u \cdot \mathsf{goto}(l_b)$ and $y = m_p \cdot \mathsf{ret} \cdot l_b \cdot m_b \cdot v$ makes the first conjunct true. The second conjunct is established by a fixpoint induction for $\mu\mathcal{P}$:

Admissibility is straightforward and the base case follows easily from the fact that $\bot \ ; \mathsf{wrp}_A(e, f, g) = \bot$. For the induction step assume that X is given such that for all e, f, g

$$\mathsf{wrp}_A(x, l_p \cdot y, g \cdot \langle e, f\rangle) \geq X \ ; \mathsf{wrp}_A(e, f, g) \ . \qquad (3)$$

Now, $\mathsf{fit}(\eta[p \mapsto X], \delta[p \mapsto l_p], u \cdot m \cdot v)$ holds: for $q \neq p$ we can argue as above and for $q = p$ this follows from (3). Thus, by using the induction hypothesis of the structural induction applied to π_p we can calculate as follows for arbitrarily given e, f, g:

$$\begin{aligned} & \mathsf{wrp}_A(x, l_p \cdot y, g \cdot \langle e, f\rangle) \\ \geq & \quad \{\text{Law [Label-wrp] and unfolding of } y\} \\ & \mathsf{wrp}_A(x \cdot l_p, m_p \cdot \mathsf{ret} \cdot l_b \cdot m_b \cdot v, g \cdot \langle e, f\rangle) \\ \geq & \quad \{\text{Induction hypothesis applied to } \pi_p\} \\ & \mathsf{wrp}_A^{\eta[p\mapsto X]}(\pi_p) \ ; \mathsf{wrp}_A(x \cdot l_p \cdot m_p, \mathsf{ret} \cdot l_b \cdot m_b \cdot v, g \cdot \langle e, f\rangle) \\ \geq & \quad \{\text{Definition of } \mathcal{P} \text{ and law [Ret-wrp]}\} \\ & \mathcal{P}(X) \ ; \mathsf{wrp}_A(e, f, g) \ . \end{aligned}$$

This completes the fixpoint induction. □

9 Conclusion

Two interweaved aspects motivated us to write the present paper. First of all we wanted to prove correct translation of a language with procedures to abstract assembler code; not just somehow or other but in an elegant and comprehensible manner. Algebraic calculations with predicate transformers turned out to be an adequate means for languages without procedures (see, e.g., [9]), so we decided to apply this technique in the extended scenario, too. The second stimulus is due to [10], where we proposed to employ wrp-semantics in compiler proofs. Real processors are always limited by their finite memory and a realistic notion of translation correctness must be prepared to deal with errors resulting from this limitation. We hope that the current paper demonstrates convincingly that wrp-based reasoning can cope with finite machines without burdening the verification.

The target language studied in this paper provides an adequate level of abstraction for further refinement down to actual binary machine code. The instructions may be considered as 'macros' for instructions of a more concrete assembler or machine language. Labels facilitate this, as they allow to describe destination of jumps independently from the length of code. An interesting aspect of our proof is that it shows how to handle the transition from tree-structured source programs to 'flat' target code. For this purpose we established a stronger claim that involves a universal quantification over syntactic target program contexts. This should be contrasted to the use of a tree-structured assembler language in [11] where translation correctness for a While-language without procedures is investigated. The proof in [11] does not immediately generalize to flat code.

Future work includes studying the relativized version of preservation of partial correctness ($\infty \in A$). In this case, semantics of recursive constructs is given by greatest rather than least fixpoints. As a consequence, fixpoint reasoning based on the fixpoints in the source language does not seem to work. We intend to use a fixpoint characterization of the target language's semantics instead. We also are working on concretizing from the abstract assembler language towards a realistic processor.

References

1. E. Börger and I. Durdanović. Correctness of compiling Occam to transputer code. *The Computer Journal*, 39(1), 1996.
2. L. M. Chirica and D. F. Martin. Towards compiler implementation correctness proofs. *ACM TOPLAS*, 8(2):185–214, April 1986.
3. E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics.* Texts and Monographs in Computer Science. Springer-Verlag, 1990.
4. J. D. Guttman, J. D. Ramsdell, and M. Wand. VLISP: A verified implementation of Scheme. *Lisp and Symbolic Computation*, 8:5–32, 1995.
5. C. A. R. Hoare, I. J. Hayes, H. Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorenson, J. M. Spivey, and B. A. Sufrin. Laws of programming. *Communications of the ACM*, 30(8):672–687, August 1987.
6. C. A. R. Hoare, H. Jifeng, and A. Sampaio. Normal form approach to compiler design. *Acta Informatica*, 30:701–739, 1993.
7. H. Langmaack. Software engineering for certification of systems: specification, implementation, and compiler correctness (in German). *Informationstechnik und Technische Informatik*, 39(3):41–47, 1997.
8. J. S. Moore. *Piton, A Mechanically Verified Assembly-Level Language.* Kluwer Academic Publishers, 1996.
9. M. Müller-Olm. *Modular Compiler Verification: A Refinement-Algebraic Approach Advocating Stepwise Abstraction*, LNCS 1283. Springer-Verlag, 1997.
10. M. Müller-Olm and A. Wolf. On excusable and inexcusable failures: towards an adequate notion of translation correctness. In *FM '99*, LNCS 1709, pp. 1107–1127. Springer-Verlag, 1999.
11. H. R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction.* Wiley, 1992.
12. T. S. Norvell. Machine code programs are predicates too. In *6th Refinement Workshop*, Workshops in Computing. Springer-Verlag and British Computer Society, 1994.
13. E. Pofahl. Methods used for inspecting safety relevant software. In *High Integrity Programmable Electronics*, pages 13–14. Dagstuhl-Sem.-Rep. 107, 1995.

A Kleene Analysis of Mobile Ambients

Flemming Nielson[1], Hanne Riis Nielson[1], and Mooly Sagiv[2]

[1] Aarhus University
[2] Tel Aviv University

Abstract. We show how a program analysis technique originally developed for *C-like pointer structures* can be adapted to analyse the hierarchical structure of processes in the *ambient calculus*. The technique is based on modeling the semantics of the language in a two-valued logic; by reinterpreting the logical formulae in Kleene's *three-valued logic* we obtain an analysis allowing us to reason about *may* as well as *must* properties. The correctness of the approach follows from a general Embedding Theorem for Kleene's logic; furthermore embeddings allow us to reduce the size of structures so as to control the time and space complexity of the analysis.

1 Introduction

Mobile ambients. The ambient calculus is a prototype web-language that allows processes (in the form of mobile ambients) to move inside a hierarchy of administrative domains (also in the form of mobile ambients); since the processes may continue to execute during their movement this notion of mobility extends that found in Java where only passive code in the form of applets may be moved. Mobile ambients were introduced in [1] and have been studied in [2,3,4,9,13,16,17]. The calculus is patterned after the π-calculus but focuses on named ambients and their movement rather than on channel-based communication; indeed, already the communication-free fragment of the calculus is very powerful (and in particular Turing complete); we review it in Section 2.

Since processes may evolve when moving around it is hard to predict which ambients may turn up inside what other ambients. In this paper we present an analysis that allows us to validate whether all executions satisfy properties like:

- Is there always exactly one copy of the ambient p?
- Is p always inside at most one of the ambients r_1, r_2 and r_3?

Kleene's three-valued logic. In [18] Kleene's three-valued logic is used to obtain safe approximations to the shape of dynamically evolving C-like pointer structures. From a *programming language point of view*, the setting of the present paper is vastly different. In contrast to traditional imperative languages the ambient calculus has no separation of program and data, furthermore non-determinism and concurrency are crucial ingredients and hence the notion of program point is demoted. The central operations on C-like pointers are assignments, which are executed one at a time and with a local effect on the heap;

G. Smolka (Ed.): ESOP/ETAPS 2000, LNCS 1782, pp. 305–319, 2000.

this is in contrast to the reductions of the ambient calculus which may happen in a number of not a priori known contexts – thus the overall effect is hard to predict. We present an overview of Kleene's three-valued logic in Section 3.

Predicate logic as a meta-language. Our overall approach is a *meta-language approach* as known for example from the denotational approach to program analysis [14]. However, here we are based on a *predicate logic* with equality and appropriate non-logical predicates as well as operations for their transitive closure; the choice of non-logical predicates is determined by the programming language or calculus at hand. To deal with equality we rely on the presence of a special unary *summary predicate* indicating whether or not an individual represents one or more entities. The important point is that the logic must be powerful enough to express both

- the properties of the configurations that we are interested in, and
- a postcondition semantics for transitions between configurations.

From a *process algebra point of view*, our representation of ambients (presented in Section 4) is rather low-level as it operates over structures containing a universe of explicit individuals; the non-logical predicates are then used to encode mobile ambients within these structures. The benefit of using sets of individuals, over the more standard formulation using multisets of subambients, is that it directly allows us to use the logical formulae.

Static analysis. The aim of the analysis is to identify certain invariant properties that hold for all executions of the system; from the *process algebra point of view*, the invariants are akin to types and they represent the sets of ambients that can arise. Since the set of ambient structures may be infinite, the analysis needs to perform an abstraction to remain tractable.

For a moment let us assume that we continue interpreting the specification in a two-valued logic. From the classical *program analysis point of view*, the maxim that "program analysis always errs on the safe side" amounts to saying that the truth values *false* and *true* of the concrete world are being replaced by 0 (for *false* or *cannot*) and a value 1/2 (for *possibly true* or *may*) in the abstract world. As an example, if a reachability analysis says that a given value (or ambient) cannot reach a given point (or ambient) then indeed it cannot, but if the reachability analysis says that it might then it may be a "false positive" due to the imprecision of the analysis.

The Embedding Theorem. The power of our approach comes from the ability to reinterpret the transition formulae over Kleene's three-valued logic. Here we have the truth values 0 (for *false* or *cannot*), 1 (for *true* or *must*) and 1/2 (for *possibly true* or *may*) [18]. The benefit of this is that we get a distinction between *may* and *must* properties for free! Returning to the reachability example we are now able to express certain cases where a value (or ambient) definitely reaches a given point.

It is straightforward to reinterpret the specification of the semantics over structures that allow all three truth values (see Section 5). The correctness of the approximate analysis with respect to the semantics follows using a general Embedding Theorem that shows that the interpretation in Kleene's three-valued logic is conservative over the ordinary two-valued interpretation. Termination is guaranteed thanks to our techniques for restricting attention to a bounded universe of individuals and for combining certain structures into one (thereby possibly introducing more 1/2's); these techniques generally work by allowing us to compress structures so that fewer individuals (perhaps only boundedly many) are needed and thereby allowing us to control the time and space complexity of our analysis.

2 The Ambient Calculus

Syntax and informal semantics. We shall study the communication-free subset of the ambient calculus [1,2] and for simplicity of presentation we shall follow [2] in dispensing with local names. Given a supply of *ambient names* $n \in N$ we define the syntax of *processes* $P \in \mathbf{Proc}$ and *capabilities* $M \in \mathbf{Cap}$:

$$\begin{aligned} P &::= \mathtt{0} \mid P \parallel P' \mid !P \mid n[\,P\,] \mid M.\,P \\ M &::= \mathtt{in}\ n \mid \mathtt{out}\ n \mid \mathtt{open}\ n \end{aligned}$$

The first constructs are well-known from the π-calculus. The process $\mathtt{0}$ is the inactive process; as usual we shall omit trailing $\mathtt{0}$'s. We write $P \parallel P'$ for the parallel composition of the two processes P and P' and we write $!P$ for the replicated process that can evolve into any number of parallel occurrences of P.

The remaining constructs are specific to the ambient calculus. The construct $n[\,P\,]$ encapsulates the process P in the ambient n. In the basic ambient calculus a process can perform three operations: it can move into a sibling ambient using the $\mathtt{in}\ n$ capability, it can move out of the parent ambient using the $\mathtt{out}\ n$ capability or it can dissolve a sibling ambient using the $\mathtt{open}\ n$ capability. These operations are illustrated pictorially in Fig. 1 where we draw the processes as trees: the nodes of the trees are labelled with names and capabilities and the subtrees represent parallel processes "inside" the parent. The figure expresses that when a process matches the upper part of one of the rules then it can be replaced by a process of the form specified by the lower part. The reduction can take place in a subprocess occurring deeply inside several ambients; however, capabilities always have to be executed sequentially.

Example 1. Throughout the paper we shall consider the following example:

$$\begin{aligned} p[\,\mathtt{in}\ r_1 \parallel !\ \mathtt{open}\ r\,] &\parallel r_1[\ !\ r[\,\mathtt{in}\ p.\ \mathtt{out}\ r_1.\ \mathtt{in}\ r_2\,]\,] \\ &\parallel r_2[\ !\ r[\,\mathtt{in}\ p.\ \mathtt{out}\ r_2.\ \mathtt{in}\ r_3\,] \parallel !\ r[\,\mathtt{in}\ p.\ \mathtt{out}\ r_2.\ \mathtt{in}\ r_1\,]\,] \\ &\parallel r_3[\,] \end{aligned}$$

It illustrates how a package p is first passed to the site r_1, then to r_2 from which it is either passed to r_3 or back to r_1.

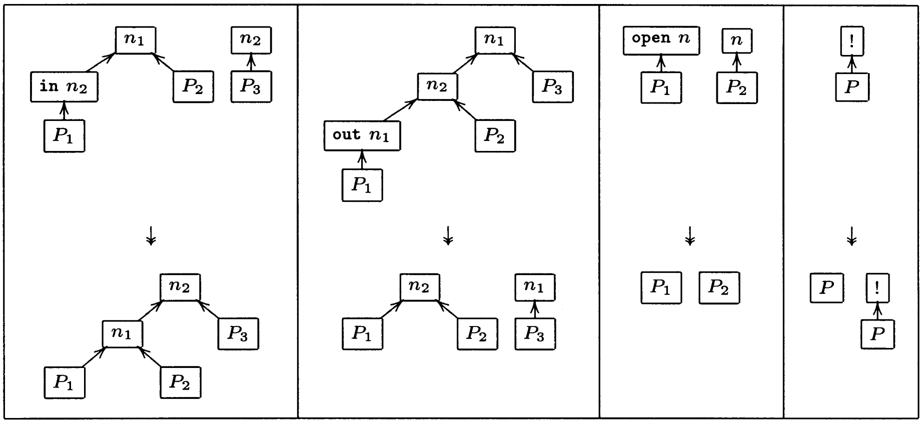


Fig. 1. Semantics of capabilities and replication.

Formal semantics. Formally the semantics is specified by a structural congruence relation $P \equiv Q$ allowing us to rearrange the appearance of processes (e.g. corresponding to reordering the order of the descendants of a node in a tree) and a reduction relation $P \twoheadrightarrow Q$ modelling the actual computation; the only deviation from [1] is that the semantics of replication is part of the transition relation rather than the congruence relation (see Fig. 1).

3 A Primer on Three-valued Logic

Syntax. It will be convenient to use a slight generalization of the logic used in [18]. Let $Pr[k]$ denote the set of predicate symbols of arity k and let $Pr = \bigcup_k Pr[k]$ be their *finite* union. We shall write $=$ for the equality predicate and furthermore we shall assume that $Pr[1]$ contains a special predicate sm. Here sm stands for "summary-predicate" and we shall later interpret it as meaning that its argument might represent multiple individuals. Without loss of generality we exclude constant and function symbols from our logic; instead we encode constant symbols as unary predicates and n-ary functions as $n+1$-ary predicates.

We write formulae over Pr using the logical connectives $\vee$, $\neg$ and the quantifier $\exists$; the formal syntax is:

$$\begin{array}{lll}
\varphi ::= & v_1 = v_2 & \text{equality on individuals} \\
\mid & p(v_1, v_2, \ldots, v_k) & \text{predicate value, } p \in Pr[k] \\
\mid & R^k(v_1, v_2, \ldots, v_k) & \text{application of second order free variable} \\
\mid & p^+(v_1, v_2) & \text{transitive closure of a relation } p,\ p \in Pr[2] \\
\mid & \varphi_1 \vee \varphi_2 & \text{disjunction} \\
\mid & \neg\varphi_1 & \text{negation} \\
\mid & \exists v : \varphi & \text{(first order) existential quantification}
\end{array}$$

Capital letters of the form R^k are used for (second-order) relations of arity k. We also use several shorthands: $\forall v : \varphi$ stands for $\neg\exists v : \neg\varphi$ and $\varphi_1 \wedge \varphi_2$ stands for

Table 1. Kleene's three-valued interpretation of the propositional operators.

$\wedge$	0	1	1/2
0	0	0	0
1	0	1	1/2
1/2	0	1/2	1/2

$\vee$	0	1	1/2
0	0	1	1/2
1	1	1	1
1/2	1/2	1	1/2

	$\neg$
0	1
1	0
1/2	1/2

$\neg(\neg\varphi_1 \vee \neg\varphi_2)$. The above shorthands are useful since three-valued logic formulae obey De-Morgan laws. Also $\varphi_1 \implies \varphi_2$ stands for $(\neg\varphi_1 \vee \varphi_2)$ and $v_1 \neq v_2$ stands for $\neg(v_1 = v_2)$. Finally, we assume the standard associativity and precedence rules.

Semantics. A *two-valued interpretation* of the language of formulae over Pr is a structure $T = \langle U, \iota\rangle_2$, where U is a set of *individuals* and ι maps each predicate symbol p of arity k to a truth-valued function:

$$\iota \colon Pr[k] \to U^k \to \{0, 1\}.$$

A *three-valued interpretation* is then a structure $T = \langle U, \iota\rangle_3$, where now ι maps each predicate symbol p of arity k to a truth-valued function:

$$\iota \colon Pr[k] \to U^k \to \{0, 1, 1/2\}.$$

We use Kleene's three-valued semantics which operates with the three values: 0, 1 and 1/2. The values 0 and 1 are called *definite values* and 1/2 is an *indefinite value.* The informal semantics of this logic is given in Table 1 where 1/2 represents situations were the result may be either true or false. Alternatively, think of 1 as representing $\{\mathit{true}\}$, 0 as representing $\{\mathit{false}\}$, and 1/2 as representing $\{\mathit{false}, \mathit{true}\}$. In the propositional case, our presentation of three-valued logics here follows [7, Chapter 8]. We shall omit the subscripts $_2$ and $_3$ when it is clear from the context whether we are in a two-valued or a three-valued world.

The semantics is rather standard given Table 1; due to space limitations we dispense with the formalisation. There are, however, one slightly non-standard aspect, namely the meaning of equality (denoted by the symbol '='). This comes from our desire to compactly represent multiple concrete elements with the same "abstract element". Therefore, the meaning of the predicate = is defined in terms of the unary summary predicate, *sm*, that expresses that an individual represents more than one entity, and the equality, =, upon individuals:

- Non-identical individuals are not equal: $u_1 = u_2$ yields 0 if $u_1 \neq u_2$.
- A non-summary individual is equal to itself: $u = u$ yields 1 if $sm(u) = 0$.
- A summary individual may be equal to itself: $u = u$ yields 1/2 if $sm(u) = 1/2$.

Here we exploit the fact that $sm(u)$ is never allowed to give 1. This will be made more precise when defining the notion of plain (two-valued) and blurred (three-valued) structures in Section 4.

Since we are interested in program analysis it is important to observe that there is an *information ordering* $\sqsubseteq$ where $l_1 \sqsubseteq l_2$ denotes that l_1 has more definite

Table 2. The intended meaning of the predicate symbols.

predicate	**intended meaning**
$pa(v_1, v_2)$	is v_2 an immediate parent of v_1?
$m(v)$	does v denote an occurrence of an ambient named m?
$in\ m(v)$	does v denote an occurrence of an action $\texttt{in}\ m$?
$out\ m(v)$	does v denote an occurrence of an action $\texttt{out}\ m$?
$open\ m(v)$	does v denote an occurrence of an action $\texttt{open}\ m$?
$!(v)$	does v denote an occurrence of a replicator operation $\texttt{!}$?
$sm(v)$	does v represent more than one element?

information than l_2; formally, $l_1 \sqsubseteq l_2$ if and only if $l_1 = l_2$ or $l_2 = 1/2$. We write $\sqcup$ for the join operation with respect to $\sqsubseteq$. Viewing 0 as meaning $\{false\}$ etc. the information ordering coincides with the subset ordering and $\sqcup$ with set-union. It is important to point out that Kleene's logic is monotonic in this order.

4 The Abstract Domain

Motivation. The two ambients $n[\,m[\,\texttt{0}\,]\,]$ and $n[\,m[\,\texttt{0}\,] \| m[\,\texttt{0}\,]\,]$ are distinct because the former has only one occurrence of m inside n whereas the latter has two. In other words, the collection of constituents of an ambient denote a *multiset* rather than a *set*.

To facilitate the use of classical notions from logic we want to view the collection of constituents as a set rather than as a multiset. We do this by introducing a set of individuals, so that the collection of constituents simply are a set of individuals. Informally, the above ambients will be represented as $(u_1 : n)[\,(u_2 : m)[\,\texttt{0}\,]\,]$ and $(u_1 : n)[\,(u_2 : m)[\,\texttt{0}\,] \| (u_3 : m)[\,\texttt{0}\,]\,]$, respectively.

Once we have introduced the notion of individuals we are ready to model ambients by structures of the kind already mentioned in Section 3 and defined formally below. These structures are obtained by fixing the set of predicate symbols so as to be able to represent ambients; we shall use the predicates shown in Table 2. In particular, there is a binary relation pa to represent the parent relation between individuals, and a number of unary relation symbols to represent the ambient information associated with individuals. Returning to the two ambients above we have that

$$n[\,m[\,\texttt{0}\,] \| m[\,\texttt{0}\,]\,] \text{ yields } \iota(pa)(u, u') = \begin{cases} 1 & \text{if } u \in \{u_2, u_3\} \wedge u' = u_1 \\ 0 & \text{otherwise} \end{cases}$$

and similarly for $n[\,m[\,\texttt{0}\,]\,]$.

Plain and blurred structures. We shall first fix the set of predicates to be used in the logic. Let N be a finite and non-empty set of ambient names and let $Pr = Pr[1] \cup Pr[2]$ be given by the following sets of predicates:

$$Pr[1] = \{sm\} \cup \{!\} \cup \{in\ \ m, out\ \ m, open\ \ m \mid m \in N\} \cup \{m \mid m \in N\}$$
$$Pr[2] = \{pa\}$$

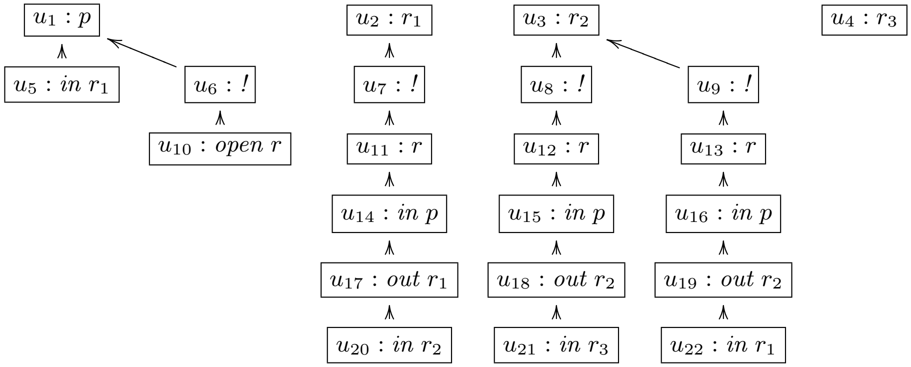


Fig. 2. A plain (two-valued) structure for the running example.

A *blurred structure* is then a three-valued interpretation $T = \langle U, \iota \rangle_3$ in the sense of Section 3 that satisfies the following conditions:

- The set U is countably infinite and $\forall u \in U : \iota(sm)(u) \neq 1$.
- The set U_{ac} defined below is finite:

$$U_{ac} = \{u \in U \mid \exists p \in Pr[1] \setminus \{sm\} : \iota(p)(u) \neq 0 \ \vee \\ \exists u' \in U : (\iota(pa)(u, u') \neq 0 \ \vee \ \iota(pa)(u', u) \neq 0)\}$$

A *plain structure* $S = \langle U, \iota \rangle_2$ is a two-level structure satisfying the above conditions; hence the unary predicate *sm* maps all individuals to 0. Plain structures suffice for representing ambients precisely. Blurred structures are only needed in order to obtain a computable analysis.

Example 2. Fig. 2 shows the plain structure that corresponds to the program of Example 1. Directed edges represent the parent relation and individuals are annotated with those unary predicates that give the value 1.

Fig. 3 shows a blurred structure for the same program. It is obtained by merging individuals from Fig. 2 that satisfy the same unary predicates. As an example the individuals u_5 ad u_{22} in Fig. 2 are merged into a summary individual $u_{5,22}$ in Fig. 3; this individual is now surrounded by a dotted box since its *sm* value is 1/2. Also, it has two outgoing dotted edges which describe the two potential places in the ambient hierarchy were the capability can occur; dotted edges means that the *pa* predicate evaluates to 1/2.

Representations of ambients. Table 3 defines a one-to-one (but not necessarily onto) mapping $\widehat{\cdot}$ from mobile ambients into plain structures. It makes use of the operation *empty* that returns a structure $\langle U, \iota \rangle$ where all predicates are interpreted so that they yield 0. The operation $new(p, \langle U, \iota \rangle)$ (for $p \in Pr[1]$) returns a structure $\langle U', \iota' \rangle$ that is as $\langle U, \iota \rangle$ except that now U'_{ac} contains an

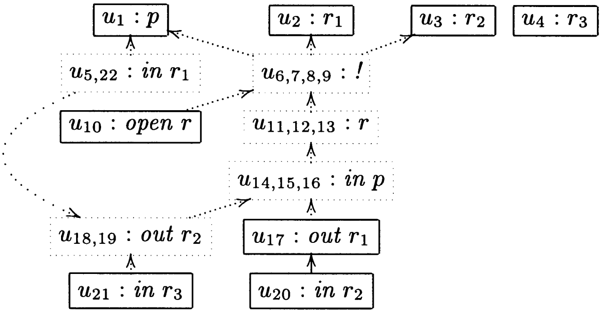


Fig. 3. A blurred (three-valued) structure for the running example.

Table 3. The mapping $\widehat{\cdot}$ from ambients to structures.

$\widehat{0} = \mathit{empty}$	$\widehat{\mathtt{in}\ m.P} = \mathit{new}(\mathit{in}\ m, \widehat{P})$
$\widehat{\mathtt{m}[P]} = \mathit{new}(m, \widehat{P})$	$\widehat{\mathtt{out}\ m.P} = \mathit{new}(\mathit{out}\ m, \widehat{P})$
$\widehat{P_1 [\![P_2} = \widehat{P_1} \uplus \widehat{P_2}$	$\widehat{\mathtt{open}\ m.P} = \mathit{new}(\mathit{open}\ m, \widehat{P})$
$\widehat{!P} = \mathit{new}(!, \widehat{P})$	

additional element r not in U_{ac}; the predicate p is set to 1 on r and all other predicates involving r are set to 0. Some of the individuals r' of U will serve as "roots" (of subprocesses represented by the structure) and the predicate pa is set to 1 on the pairs (r', r) and 0 elsewhere. We shall not formalise the concept of "roots" but only mention that the individual r will be the "root" of the new structure. Finally, the operation $\langle U, \iota\rangle \uplus \langle U', \iota'\rangle$ returns a structure $\langle U'', \iota''\rangle$ where $U'' = U \cup U'$ and we take case to rename the individuals of U and U' such that $U \cap U' = \emptyset$; the "roots" are the union of those of the two structures. Fig. 2 shows the result of applying $\widehat{\cdot}$ to the program of Example 1.

Embeddings. Next, we define an embedding order on structures.

Definition 1. *Let $T = \langle U, \iota\rangle_\kappa$ and $T' = \langle U', \iota'\rangle_3$ be two structures (for κ being 2 or 3) and let $f\colon U \to U'$ be a surjective function. We say that f embeds T in T' (written $T \sqsubseteq^f T'$) if*

- *for every $p \in Pr[k] \setminus \{sm\}$:*

$$\iota'(p)(u'_1, \ldots, u'_k) \sqsupseteq \bigsqcup_{f(u_i)=u'_i, 1\le i\le k} \iota(p)(u_1, \ldots, u_k) \tag{1}$$

- *for every $u' \in U'$:*

$$\iota'(sm)(u') \sqsupseteq \left(|\{u | f(u) = u'\}| > 1 \right) \sqcup \bigsqcup_{f(u)=u'} \iota(sm)(u) \tag{2}$$

The embedding is tight if equalities hold in (1) and (2).

We say that T can be embedded in T' (denoted by $T \sqsubseteq T'$) if there exists a function f such that $T \sqsubseteq^f T'$.

We can now establish an Embedding Theorem [18] that intuitively says:

> If T can be embedded in T', then every piece of information extracted from T' via a formula φ is a conservative approximation of the information extracted from T via φ.

The first usage of the embedding order is to define the concretization: the set of plain structures (and hence ambients) described by a blurred structure.

Definition 2. *For a blurred structure T, we denote by $\mathcal{E}(T)$ the set of plain structures S that can be embedded into T.*

If a formula φ evaluates to 1 over a blurred structure T then it is true in all plain structures $S \in \mathcal{E}(T)$; if it evaluates to 0 then it is false in all plain structures $S \in \mathcal{E}(T)$; finally, if it evaluates to 1/2 it may be true in some $S_1 \in \mathcal{E}(T)$ and false in some $S_0 \in \mathcal{E}(T)$.

Example 3. For the program of Example 1 we may be interested in the properties

$$\mathit{unique} \equiv \forall v_1, v_2 : p(v_1) \land p(v_2) \implies v_1 = v_2 \tag{3}$$

$$\begin{aligned}\mathit{position} \equiv \forall v_1, v_2, v_3 : p(v_1) \land pa^+(v_1, v_2) \land pa^+(v_1, v_3) \land ro(v_2) \land ro(v_3) \\ \implies v_2 = v_3\end{aligned} \tag{4}$$

where $ro(v) \equiv r_1(v) \lor r_2(v) \lor r_3(v)$. The formula (3) expresses that the structure only contains a single copy of the ambient p. The formula (4) expresses that the ambient p will be within at most one of the ambients r_1, r_2 and r_3. These formulae have the value 1 when evaluated on the structures of Figures 2 and 3.

Bounded Structures and Canonical Embeddings. A simple way to guarantee the termination of our analysis is by ensuring that the number of blurred structures is a priori bounded. A blurred structure $T = \langle U, \iota \rangle$ is *bounded* if for every two *different* elements $u_1, u_2 \in U_{ac}$ there exists a unary predicate $p \in Pr[1] \setminus \{sm\}$, such that $\iota(p)(u_1) \neq \iota(p)(u_2)$. Clearly, the number of different individuals in U_{ac} is then bounded by $3^{|Pr[1]|-1} = O(3^{|N|})$ and thus the number of bounded structures is finite (up to isomorphism).

Moreover, every blurred structure can be embedded into a bounded structure by "joining" the truth-values of individuals mapping into the same abstract individual. More precisely, a special kind of tight embedding, called *canonical embedding*, from structures into bounded structures is obtained by defining the embedding f to map individuals u_1 and u_2 in U_{ac} to the same individual if and only if it is not the case that there is a predicate $p \in Pr[1] \setminus \{sm\}$ such that $\iota(p)(u_1) \neq \iota(p)(u_2)$. Since the canonical embedding f is uniquely defined on T (up to isomorphism) we denote $f(T)$ simply as $\lfloor T \rfloor$.

Example 4. The blurred structure in Fig. 3 is the canonical embedding of the plain structure of Fig. 2.

Table 4. Shorthand formulae used in the transition formulae.

formula	intended-meaning	formal-meaning
$nb(v)$	v is a non-blocking action	$\bigvee_{m\in N} m(v)$
$nba(v)$	all ancestors of v are non-blocking	$\forall v_1 : pa^+(v, v_1) \Rightarrow nb(v_1)$
$sib(v_1, v_2)$	v_1 and v_2 are sibling individuals	$v_1 \neq v_2 \wedge$ $((\exists v_p : pa(v_1, v_p) \wedge pa(v_2, v_p))$ $\vee \neg(\exists v_p : pa(v_1, v_p) \vee pa(v_2, v_p)))$
$ac(v)$	v is an active individual	$(\bigvee_{p\in Pr[1]\setminus\{sm\}} ac(v))$ $\vee \exists v' : (pa(v, v') \vee pa(v', v))$

5 A Simple Analysis

We now define the effect of ambient actions by formulae that compute new structures from old. These formulae are quite natural since when interpreted over plain structures they define a semantics which is equivalent to the one of Section 2 but when interpreted over blurred structures, they are conservative due to the Embedding Theorem. Restricting our attention to bounded structures thus gives us a conservative and efficient analysis of ambients.

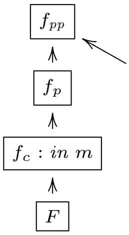
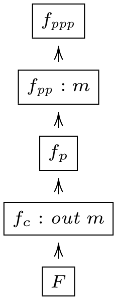
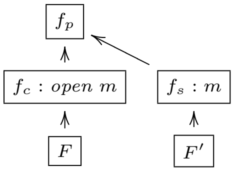
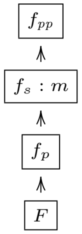
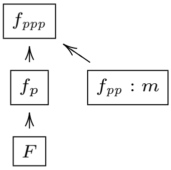
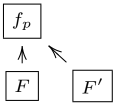
Capability actions. Table 5 defines the effect of capability actions using the shorthand formulae defined in Table 4. The semantics of Table 5 is formally defined in Definition 3. Informally, an action is characterized by the following kinds of information:

- The *condition* in which the action is enabled. It is specified as a formula with free logical variables $f_c, f_1, f_2, \ldots, f_n$ ranging over individuals. The formal variable f_c denotes the current individual and the rest of $f_1, f_2, \ldots, f_n$ denote surrounding individuals. Whenever an action is enabled, it binds the free logical variables to actual individuals in the structure. Our operational semantics is non-deterministic in the sense that many actions can be enabled simultaneously and one of them is chosen for execution.
- Enabled actions create a new structure where the interpretations of every predicate $p \in Pr[k]$ is determined by evaluating a formula $\varphi_p(v_1, v_2, \ldots, v_k)$ which may use $v_1, v_2, \ldots, v_k$ and $f_c, f_1, f_2, \ldots, f_n$ as well as all $p \in Pr$.

For simplicity, our semantics does not deal with the removal of individuals (and hence our structures use an infinite set of individuals).

Consider the treatment of the `in` m action in Table 5. It is enabled when an individual f_c is non-blocking (i.e. when there are no capabilities or replication operators on a path to a root of the structure), it denotes an `in` m action, it has a parent f_p, with a sibling individual f_s which denotes an ambient named m. Next the enabled `in` m action creates a parent relation, where f_p is connected to f_s, predecessors of f_c are connected to f_p, and individuals which are not emanating

Table 5. The structure update formulae for executing capability actions.

Action	`in` m	`out` m	`open` m
Cond. on T	$c_{in\ m}(f_c, f_p, f_s) \equiv nba(f_c)\wedge pa(f_c, f_p) \wedge sib(f_p, f_s)\wedge in\ m(f_c) \wedge m(f_s)$	$c_{out\ m}(f_c, f_p, f_{pp}) \equiv nba(f_c)\wedge pa(f_c, f_p) \wedge pa(f_p, f_{pp})\wedge out\ m(f_c) \wedge m(f_{pp})$	$c_{open\ m}(f_c, f_s) \equiv nba(f_c)\wedge sib(f_c, f_s)\wedge open\ m(f_c) \wedge m(f_s)$
Diag. of T			
$\varphi_{pa}(v_1,v_2)$	$(v_1 = f_p \wedge v_2 = f_s)\vee (pa(v_1, f_c) \wedge v_2 = f_p)\vee (pa(v_1, v_2) \wedge v_1 \neq f_c \wedge v_1 \neq f_p \wedge v_2 \neq f_c)$	$(v_1 = f_p \wedge pa(f_{pp}, v_2))\vee (pa(v_1, f_c) \wedge v_2 = f_p)\vee (pa(v_1, v_2) \wedge v_1 \neq f_c \wedge v_1 \neq f_p \wedge v_2 \neq f_c)$	$(pa(v_1, f_c) \wedge pa(f_c, v_2))\vee (pa(v_1, f_s) \wedge pa(f_s, v_2))\vee (pa(v_1, v_2) \wedge v_1 \neq f_c \wedge v_2 \neq f_c \wedge v_1 \neq f_s \wedge v_2 \neq f_s)$
$p \in Pr[1]$ $\varphi_p(v)$	$p(v) \wedge v \neq f_c$		$p(v) \wedge v \neq f_c \wedge v \neq f_s$
Diag. of T'			

from f_c or f_p and not entering f_c are not changed. On plain structures this operation exactly emulates the operational semantics (defined informally in Fig. 1). However, on blurred structure it can yield indefinite values as demonstrated in the blurred structure in Table 4, which is obtained by performing the action `in` r_1 on the blurred structure shown in Fig. 3.

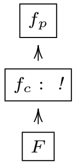
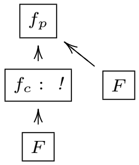
Formally, the meaning of capability actions is defined as follows:

Definition 3. *We say that a κ-valued structure $T = \langle U, \iota\rangle_\kappa$ rewrites into a structure $T' = \langle U', \iota'\rangle_\kappa$ (denoted by $T \overset{\kappa}{\to}_M T'$) where $M \in \{$**in** m, **out** m, **open** $m \mid m \in N\}$ if there exists an assignment Z such that $[\![c_M(f_c, f_1, f_2, \ldots, f_n)]\!]^T_\kappa(Z) \neq 0$ where the formula $c_M(f_c, f_1, f_2, \ldots, f_n)$ is defined in the first row of Table 5, and for every $p \in Pr[k]$ and $u_1, \ldots, u_k \in U'$,*

$$\iota'(p)(u_1, \ldots, u_k) = [\![\varphi_p(v_1, v_2, \ldots, v_k)]\!]^T_\kappa(Z[v_1 \mapsto u_1, v_2 \mapsto u_2, \ldots, v_k \mapsto u_k])$$

where $\varphi_p(v_1, \cdots, v_k)$ is the formula for p given in Table 5.

Table 6. The structure update formulae for executing the replication operation.

Action	$c_!(f_c, I^2) \equiv nba(f_c) \wedge !(f_c) \wedge$ $\forall v_1 : pa^+(v_1, f_c) \implies \exists v_2 : (\neg ac(v_2) \wedge I^2(v_1, v_2) \wedge (\forall v_3 : I^2(v_1, v_3) \implies v_2 = v_3))$
Diagram of T	
$\varphi_{pa}(v_1, v_2)$	$pa(v_1, v_2) \vee (\exists v_1', v_2' : pa(v_1', v_2') \wedge I^2(v_1', v_1) \wedge I^2(v_2', v_2))$ $\vee (\exists v_1' : pa(v_1', f_c) \wedge I^2(v_1', v_1) \wedge pa(f_c, v_2))$
$p \in Pr[1] : \varphi_p(v)$	$p(v) \vee \exists v' : p(v') \wedge I^2(v', v)$
Diagram of T'	

After reduction of in r_1 at $f_c = u_{5,22}$, $f_p = u_1$, $f_s = u_2$:

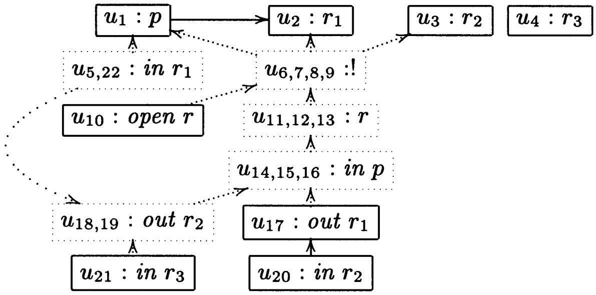


Fig. 4. The bounded structures arising in analysis after the first iteration.

Replication actions. In Table 6 we define the meaning of the replication action; we express this operation in logic by using a second order relation I^2 that creates new isomorphic individuals for the replicated ambients:

- The *condition* under which the operation is enabled: that the current individual f_c is a non-blocking replication operation.
- The update formula φ. Here we use an extra preset relation $I^2(v_1, v_2)$ which is set to true if v_2 is a new instance of an individual v_1. This relation is set before the formulae $\varphi_p(v_1, v_2, \ldots, v_k)$ is evaluated. The formulae $\varphi_p(v_1, v_2, \ldots, v_k)$ uses $v_1, v_2, \ldots, v_k$, f_c, and the new relation I^2.

On plain structures the replication operation exactly emulates the operational semantics. However, on blurred structure it can yield indefinite values and even more when the resultant structure is converted into a bounded one.

Formally, replication is handled as follows:

Definition 4. *We say that a κ-valued structure $T = \langle U, \iota\rangle$ rewrites into a structure $T' = \langle U', \iota'\rangle$ (denoted by $T \overset{\kappa}{\to}_{!} T'$) if there exists an assignment Z such that $[\![c_{!}(f_c, I^2)]\!]^T_\kappa(Z) \neq 0$ where the formula $c_{!}(f_c, I^2)$ is defined in the first row of Table 6, and for every $p \in Pr[k]$ and $u_1, u_2, \ldots, u_k \in U'$,*

$$\iota'(p)(u_1, u_2, \ldots, u_k) = [\![\varphi_p(v_1, v_2, \ldots, v_k)]\!]^T_\kappa(Z[v_1 \mapsto u_1, v_2 \mapsto u_2, \ldots, v_k \mapsto u_k])$$

where $\varphi_p(v_1, \cdots, v_k)$ is the formula for p given in Table 6.

Finally, we can formally define the analysis (or abstract semantics):

Definition 5. *We say that a κ-valued structure $T = \langle U, \iota\rangle$ rewrites into a structure $T' = \langle U, \iota'\rangle$ (denoted by $T \overset{\kappa}{\to} T'$) if either $T \overset{\kappa}{\to}_M T'$ or $T \overset{\kappa}{\to}_{!} T'$. We say that T canonically rewrites into T' (denoted by $T \overset{\lfloor 3\rfloor}{\to} T'$) when $T = \lfloor T\rfloor$ and there exists T'' such that $T' = \lfloor T''\rfloor$ and $T \overset{3}{\to} T''$. We denote by $\overset{3}{\to}^*$ the reflexive transitive closure of $\overset{3}{\to}$ and similarly for $\overset{2}{\to}^*$ and $\overset{\lfloor 3\rfloor}{\to}^*$.*

Properties of the Abstract Semantics. The set of bounded structures

$$An_T \overset{\lfloor 3\rfloor}{\to}^* T'\} \tag{5}$$

is finite and can be computed iteratively in a straightforward manner.

We can show that the semantics of processes, $\widehat{P} \twoheadrightarrow^* \widehat{Q}$, is correctly modelled by our plain rewrite relation, $\widehat{P} \overset{2}{\to}^* \widehat{Q}$. It then follows from the Embedding Theorem that the semantics of processes is correctly modelled by our blurred rewrite relation:

$$\text{Whenever } \; P \twoheadrightarrow^* Q \; \text{ we have } \; \exists T \in An_{\lfloor \widehat{P}\rfloor} : \widehat{Q} \sqsubseteq T. \tag{6}$$

Thus, we can verify safety properties of ambients by evaluating formulae against blurred structures in An_T. Of course, An_T may also include superfluous structures leading to imprecise results.

6 Conclusion

So far we have presented a very naive analysis of ambients (in the spirit of the shape analysis algorithm in [5, Section 3]). The motivation was to show the benefit of three-valued logics. We have implemented the analysis using the TVLA system [12] but in its current form it is too costly and too imprecise to give informative answers to the questions asked of the running example.

To get an efficient and sufficiently precise analysis we have refined the analysis by using two techniques already mentioned in [18]. One is to maintain finer distinctions on blurred structures based on values of so-called *instrumentation predicates* and the other is to control the complexity by *joining structures.*

Table 7. Definition of instrumentation predicates.

action	**defining formula**
`in` m	for each ambient name z : $\mathit{inside}[z,m] = \mathit{inside}[z,m] \vee \exists v : z(v) \wedge \mathit{pa}^*(v, f_p)$
`out` m	for each ambient name z : $\mathit{inside}[z,m] = \mathit{inside}[z,m] \wedge$ $(\ (\exists v_1, v_2 : z(v_1) \wedge \mathit{pa}^*(v_1, f_p) \wedge m(v_2) \wedge \mathit{pa}^+(f_{pp}, v_2)) \vee$ $(\exists v_1, v_2 : z(v_1) \wedge m(v_2) \wedge \mathit{pa}^+(v_1, v_2) \wedge \neg(\mathit{pa}^*(v_1, f_p) \wedge \mathit{pa}^*(f_p, v_2))) \ )$
`open` m	for each ambient name z : $\mathit{inside}[z,m] = \mathit{inside}[z,m] \wedge$ $(\ (\exists v_1, v_2 : z(v_1) \wedge \mathit{pa}^+(v_1, f_s) \wedge m(v_2) \wedge \mathit{pa}^*(f_p, v_2)) \vee$ $(\exists v_1, v_2 : z(v_1) \wedge m(v_2) \wedge \mathit{pa}^+(v_1, v_2) \wedge \neg(\mathit{pa}^*(v_1, f_s) \wedge \mathit{pa}^*(f_s, v_2))) \ )$

Instrumentation predicates. Technically, instrumentation predicates are just predicates which store some context information. For our analysis we have added two kinds of instrumentation predicates. One group of predicates simply labels the individual ambients and capabilities of the program much as in [9,16]. As an example, the initial blurred structure will now be similar to the initial plain structure displayed in Fig. 2 as now the active individuals will have distinct labels and hence will not be combined. The labels will remain unchanged throughout the analysis and when a part of a structure is copied as in the analysis of replication the new individuals inherit the labels of the original individuals. The benefit of adding these instrumentation predicates is that the analysis can better distinguish between the different routers in which the packet resides.

Another group of instrumentation predicates are designed to reflect the three questions we have asked which all are concerned about which ambients are inside which ambients. We therefore define nullary predicates$\mathit{inside}[z_1, z_2]$ for each combination of the ambient names $(z_1, z_2) \in N \times N$; it is defined by

$$\mathit{inside}[z_1, z_2] = \exists v_1, v_2 : z_1(v_1) \wedge z_2(v_2) \wedge \mathit{pa}^+(v_1, v_2)$$

and it is updated whenever one of the capabilities are executed as shown in Table 7; it is unchanged when the replication action is executed.

Joining structures. While the goal of adding instrumentation predicates is to get more precision, the goal of joining structures is to get more efficiency and as usual this means that we are going to loose precision. We therefore merge structures satisfying the same nullary instrumentation predicates.

With these modifications the system of [12] can indeed validate the two properties of Example 3. This took 192.6 CPU seconds on a Pentium 256 Mhz machine running NT 4.0 with JDK 1.2.

Acknowledgements. The running example was suggested by Luca Cardelli. Tal Lev-Ami provided the implementation discussed in Section 6.

References

1. L. Cardelli, A. D. Gordon: Mobile ambients. In *Proc. FoSSaCS'98*, vol. 1378 of LNCS, pages 140–155, Springer, 1998.
2. L. Cardelli, A. D. Gordon: Types for mobile ambients. In *Proc. POPL'99*, ACM Press, 1999.
3. L. Cardelli, G. Ghelli, A. D. Gordon: Mobility types for mobile ambients. In *Proc. ICALP'99*, LNCS, Springer, 1999.
4. L. Cardelli and A. D. Gordon: Anytime, Anywhere: Modal Logics for Mobile Ambients. In *Proc. POPL'00*, ACM Press, 2000.
5. D.R. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 296–310, ACM Press, 1990.
6. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Symp. on Princ. of Prog. Lang.*, pages 269–282, ACM Press, 1979.
7. R.L. Epstein. *The Semantic Foundations of Logic, Volume 1: Propositional Logics.* Kluwer Academic Publishers, 1990.
8. M.L. Ginsberg. Multivalued logics: A uniform approach to inference in artificial intelligence. In *Comp. Intell.*, 4:265–316, 1988.
9. R. R. Hansen, J. G. Jensen, F. Nielson, H. Riis Nielson: Abstract Interpretation of Mobile Ambients. In *Proc. SAS'99*, vol. 1694 of LNCS, Springer, 1999.
10. H. Hoddes. Three-Valued Logics: An Introduction, A Comparison of Various Logical Lexica, and Some Philosophical Remarks. In *Annals of Pure and Applied Logic*, 1989.
11. S.C. Kleene. *Introduction to Metamathematics.* North-Holland, second edition, 1987.
12. T. Lev-Ami: TVLA: a framework for Kleene based static analysis. M.Sc. thesis, Tel Aviv University, January 2000.
13. F. Levi and D. Sangiorgi: Controlling Interference in Ambients. In *Proc. POPL'00*, ACM Press, 2000.
14. F. Nielson: Semantics-Directed Program Analysis: A Tool-Maker's Perspective. In *Proc. SAS'96*, vol. 1145 of LNCS, Springer, 1996.
15. F. Nielson, H. Riis Nielson, C. L. Hankin: *Principles of Program Analysis*, Springer, 1999.
16. F. Nielson, H. Riis Nielson, R. R. Hansen, J. G. Jensen: Validating Firewalls in Mobile Ambients. In *Proc. CONCUR'99*, vol. 1664 of LNCS, Springer, 1999.
17. F. Nielson, H. Riis Nielson: Shape Analysis for Mobile Ambients. In *Proc. POPL'00*, ACM Press, 2000.
18. M. Sagiv, T. Reps, R. Wilhelm: Parametric Shape Analysis via 3-Valued Logic. In *Proc. POPL'99*, ACM Press, 1999.

A 3-Part Type Inference Engine

François Pottier

INRIA Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France.
Francois.Pottier@inria.fr

Abstract. Extending a *subtyping-constraint-based type inference* framework with *conditional constraints* and *rows* yields a powerful type inference engine. We illustrate this claim by proposing solutions to three delicate type inference problems: "accurate" pattern matchings, record concatenation, and "dynamic" messages. Until now, known solutions required significantly different techniques; our theoretical contribution is in using only a single (and simple) set of tools. On the practical side, this allows all three problems to benefit from a common set of constraint simplification techniques, leading to efficient solutions.

1 Introduction

Type inference is the task of examining a program which lacks some (or even all) type annotations, and recovering enough type information to make it acceptable by a type checker. Its original, and most obvious, application is to free the programmer from the burden of manually providing these annotations, thus making static typing a less dreary discipline. However, type inference has also seen heavy use as a simple, modular way of formulating program analyses.

This paper presents a common solution to several seemingly unrelated type inference problems, by unifying in a single type inference system several previously proposed techniques, namely: a simple framework for *subtyping-constraint-based type inference* [15], *conditional constraints* inspired by Aiken, Wimmers and Lakshman [2], and *rows* à la Rémy [18].

Constraint-Based Type Inference

Subtyping is a partial order on types, defined so that an object of a subtype may safely be supplied wherever an object of a supertype is expected. Type inference in the presence of subtyping reflects this basic principle. Every time a piece of data is passed from a producer to a consumer, the former's output type is required to be a *subtype* of the latter's input type. This requirement is explicitly recorded by creating a symbolic *subtyping constraint* between these types. Thus, each potential data flow discovered in the program yields one constraint. This fact allows viewing a constraint set as a directed approximation of the program's data flow graph – regardless of our particular definition of subtyping.

Various type inference systems based on subtyping constraints exist. One may cite works by Aiken et al. [1, 2, 5], the present author [16, 15], Trifonov

G. Smolka (Ed.): ESOP/ETAPS 2000, LNCS 1782, pp. 320–335, 2000.

and Smith [22], as well as an abstract framework by Odersky, Sulzmann and Wehr [12]. Related systems include set-based analysis [8, 6] and type inference systems based on feature constraints [9, 10].

Conditional Constraints

In most constraint-based systems, the expression `if` e_0 `then` e_1 `else` e_2 may, at best, be described by

$$\alpha_1 \leq \alpha \quad \wedge \quad \alpha_2 \leq \alpha$$

where α_i stands for e_i's type, and α stands for the whole expression's type. This amounts to stating that "e_1's (resp. e_2's) value may become the whole expression's value", regardless of the test's outcome. A more precise description – "if e_0 may evaluate to `true` (resp. `false`), then e_1's (resp e_2's) value may become the whole expression's value" – may be given using natural *conditional constraints*:

$$\texttt{true} \leq \alpha_0 \,?\, \alpha_1 \leq \alpha \quad \wedge \quad \texttt{false} \leq \alpha_0 \,?\, \alpha_2 \leq \alpha$$

Introducing tests into constraints allows keeping track of the program's *control* flow – that is, mirroring the way evaluation is affected by a test's outcome, at the level of types.

Conditional set expressions were introduced by Reynolds [21] as a means of solving set constraints involving strict type constructors and destructors. Heintze [8] uses them to formulate an analysis which ignores "dead code". He also introduces *case constraints*, which allow ignoring the effect of a branch, in a `case` construct, unless it is actually liable to be taken. Aiken, Wimmers and Lakshman [2] use *conditional types*, together with intersection types, for this purpose.

In the present paper, we suggest a single notion of *conditional constraint*, which is comparable in expressive power to the above constructs, and lends itself to a simple and efficient implementation. (A similar choice was made independently by Fähndrich [5].) We emphasize its use as a way not only of introducing *control* into types, but also of *delaying* type computations, thus introducing some "laziness" into type inference.

Rows

Designing a type system for a programming language with records, or objects, requires some way of expressing labelled products of types, where labels are field or method names. Dually, if a programming language allows manipulating structured data, then its type system shall likely require labelled sums, where labels are names of data constructors.

Rémy [18] elegantly deals with both problems at once by introducing notation to express denumerable, indexed families of types, called *rows*:

$$\rho ::= \alpha, \beta, \ldots, \varphi, \psi, \ldots \mid a : \tau;\ \rho \mid \partial\tau$$

(Here, τ ranges over types, and $a, b, \ldots$ range over indices.) An unknown row may be represented by a *row variable*, exactly as in the case of types. (By lack of symbols, we shall not syntactically distinguish regular type variables and row variables.) The term $a : \tau;\ \rho$ represents a row whose element at index a is τ, and whose other elements are given by ρ. The term $\partial\tau$ stands for a row whose element at any index is τ. These informal explanations are made precise via an equational theory:

$$a : \tau_a;\ (b : \tau_b;\ \rho) = b : \tau_b;\ (a : \tau_a;\ \rho)$$
$$\partial\tau = a : \tau;\ \partial\tau$$

For more details, we refer the reader to [18].

Rows offer a particularly straightforward way of describing operations which treat all labels (except possibly a finite number thereof) uniformly. Because every facility available at the level of types (e.g. constructors, constraints) can also be made available at the level of rows, a description of what happens at the level of a single label – written using types – can also be read as a description of the whole operation – written using rows. This interesting point will be developed further in the paper.

Putting It All Together

Our point is to show that the combination of the three concepts discussed above yields a very expressive system, which allows type inference for a number of advanced language features. Among these, "accurate" pattern matching constructs, record concatenation, and "dynamic" messages will be discussed in this paper. Our system allows performing type inference for all of these features at once. Furthermore, efficiency issues concerning constraint-based type inference systems have already been studied [5, 15]. This existing knowledge benefits our system, which may thus be used to *efficiently* perform type inference for all of the above features.

In this paper, we focus on *applications* of our type system, i.e. we show how it allows solving each of the problems mentioned above. Theoretical aspects of constraint solving are discussed in [15, 17]. Furthermore, a robust prototype implementation is publicly available [14]. We do not prove that the types given to the three problematic operations discussed in this paper are sound, but we believe this is a straightforward task.

The paper is organized as follows. Section 2 gives a brief technical overview of the type system, focusing on the notion of constrained type scheme, which should be enough to gain an understanding of the paper. Sections 3, 4, and 5 discuss type inference for "accurate" pattern matchings, record concatenation, and "dynamic" messages, respectively, within our system. Section 6 sums up our contribution, then briefly discusses future research topics. Appendix A gives some more technical details, including the system's type inference rules. Lastly, Appendix B gives several examples, which show what inferred types look like in practice.

2 System's Overview

The programming language considered throughout the paper is a call-by-value λ-calculus with let-polymorphism, i.e. essentially core ML.

$$e ::= x, y, \ldots \mid \lambda x.e \mid (e\, e) \mid X, Y, \ldots \mid \texttt{let}\ X = e\ \texttt{in}\ e$$

The type algebra needed to deal with such a core language is simple. The set of *ground terms* contains all regular trees built over $\bot$, $\top$ (with arity 0) and $\rightarrow$ (with arity 2). It is equipped with a straightforward *subtyping* relationship [15], denoted $\leq$, which makes it a lattice. It is the logical model in which subtyping constraints are interpreted.

Symbols, *type variables*, *types* and *constraints* are defined as follows:

$$\begin{array}{ll} s ::= \bot \mid \rightarrow \mid \top & v ::= \alpha, \beta, \ldots \\ \tau ::= v \mid \bot \mid \tau \rightarrow \tau \mid \top & c ::= \tau \leq \tau \\ & \quad\mid\ \ s \leq v\, ?\, \tau \leq \tau \end{array}$$

A ground substitution ϕ is a map from type variables to ground terms. A constraint of the form $\tau_1 \leq \tau_2$, which reads "τ_1 must be a subtype of τ_2", is satisfied by ϕ if and only if $\phi(\tau_1) \leq \phi(\tau_2)$. A constraint of the form $s \leq \alpha\, ?\, \tau_1 \leq \tau_2$, which reads "if α exceeds s, then τ_1 must be a subtype of τ_2", is satisfied by ϕ if and only if $s \leq_{\mathcal{S}} \text{head}(\phi(\alpha))$ implies $\phi(\tau_1) \leq \phi(\tau_2)$, where head maps a ground term to its head constructor, and $\leq_{\mathcal{S}}$ is the expected ordering over symbols. A constraint set C is satisfied by ϕ if and only if all of its elements are.

A *type scheme* is of the form

$$\sigma ::= \forall C.\, \tau$$

where τ is a type and C is a constraint set, which restricts the set of σ's ground instances. Indeed, the latter, which we call σ's *denotation*, is defined as

$$\{\tau' \,;\, \exists \phi \quad \phi \text{ satisfies } C \wedge \phi(\tau) \leq \tau'\}$$

Because *all* of a type scheme's variables are universally quantified, we will usually omit the $\forall$ quantifier and simply write "τ where C".

Of course, the type algebra given above is very much simplified. In general, the system allows defining more type constructors, separating symbols (and terms) into *kinds*, and making use of rows. (A full definition – without rows – appears in [17].) However, for presentation's sake, we will introduce these features only step by step.

The core programming language described above is also limited. To extend it, we will define new primitive operations, equipped with an operational semantics and an appropriate *type scheme*. However, no extension to the *type system* – e.g. in the form of new typing rules – will be made. This explains why we do not further describe the system itself. (Some details are given in Appendix A.) Really, all this paper is about is *writing expressive constrained type schemes*.

3 Accurate Analysis of Pattern Matchings

When faced with a pattern matching construct, most existing type inference systems adopt a simple, conservative approach: assuming that each branch may be taken, they let it contribute to the whole expression's type. A more accurate system should use types to prove that certain branches cannot be taken, and prevent them from contributing.

In this section, we describe such a system. The essential idea – introducing a conditional construct at the level of types – is due to [8, 2]. Some novelty resides in our two-step presentation, which we believe helps isolate independent concepts. First, we consider the case where only *one* data constructor exists. Then, we easily move to the general case, by enriching the type algebra with rows.

3.1 The Basic Case

We assume the language allows building and accessing tagged values.

$$e ::= \ldots \mid \texttt{Pre} \mid \texttt{Pre}^{-1}$$

A single data constructor, Pre, allows building tagged values, while the destructor $\texttt{Pre}^{-1}$ allows accessing their contents. This relationship is expressed by the following reduction rule:

$$\texttt{Pre}^{-1}\, v_1\, (\texttt{Pre}\, v_2) \quad \text{reduces to} \quad (v_1\, v_2)$$

The rule states that $\texttt{Pre}^{-1}$ first takes the tag off the value v_2, then passes it to the function v_1.

At the level of types, we introduce a (unary) variant type constructor $[\,\cdot\,]$. Also, we establish a distinction between so-called "regular types," written τ, and "field types," written ϕ.

$$\tau ::= \alpha, \beta, \gamma, \ldots \mid \bot \mid \top \mid \tau \rightarrow \tau \mid [\,\phi\,]$$
$$\phi ::= \varphi, \psi, \ldots \mid \texttt{Abs} \mid \texttt{Pre}\ \tau \mid \texttt{Any}$$

A subtype ordering over field types is defined straightforwardly: Abs is its least element, Any is its greatest, and Pre is a covariant type constructor.

The data constructor Pre is given the following type scheme:

$$\texttt{Pre} : \alpha \rightarrow [\,\texttt{Pre}\ \alpha\,]$$

Notice that there is no way of building a value of type $[\,\texttt{Abs}\,]$. Thus, if an expression has this type, then it must diverge. This explains our choice of names. If an expression has type $[\,\texttt{Abs}\,]$, then its value must be "absent"; if it has type $[\,\texttt{Pre}\ \tau\,]$, then some value of type τ may be "present".

The data destructor $\texttt{Pre}^{-1}$ is described as follows:

$$\begin{aligned} \texttt{Pre}^{-1} &: (\alpha \to \beta) \to [\,\varphi\,] \to \gamma \\ \text{where} \quad & \varphi \leq \texttt{Pre}\ \alpha \\ & \texttt{Pre} \leq \varphi\,?\,\beta \leq \gamma \end{aligned}$$

The conditional constraint allows $(\texttt{Pre}^{-1}\, e_1\, e_2)$ to receive type $\bot$ when e_2 has type $[\,\texttt{Abs}\,]$, reflecting the fact that $\texttt{Pre}^{-1}$ isn't invoked until e_2 produces some value. Indeed, as long as φ equals $\texttt{Abs}$, the constraint is vacuously satisfied, so γ is unconstrained and assumes its most precise value, namely $\bot$. However, as soon as $\texttt{Pre} \leq \varphi$ holds, $\beta \leq \gamma$ must be satisfied as well. Then, $\texttt{Pre}^{-1}$'s type becomes equivalent to $(\alpha \to \beta) \to [\,\texttt{Pre}\ \alpha\,] \to \beta$, which is its usual ML type.

3.2 The General Case

We now move to a language with a denumerable set of data constructors.

$$e ::= \ldots \mid K \mid K^{-1} \mid \texttt{close}$$

(We let K, L, ... stand for data constructors.) An expression may be tagged, as before, by applying a data constructor to it. Accessing tagged values becomes slightly more complex, because multiple tags exist. The semantics of the elementary data destructor, K^{-1}, is given by the following reduction rules:

$$\begin{array}{lll} K^{-1}\, v_1\, v_2\, (K\, v_3) & \text{reduces to} & (v_1\, v_3) \\ K^{-1}\, v_1\, v_2\, (L\ \ v_3) & \text{reduces to} & (v_2\, (L\, v_3)) \quad \text{when } K \neq L \end{array}$$

According to these rules, if the value v_3 carries the expected tag, then it is passed to the function v_1. Otherwise, the value – still carrying its tag – is passed to the function v_2. Lastly, a special value, $\texttt{close}$, is added to the language, but no additional reduction rule is defined for it.

How do we modify our type algebra to accommodate multiple data constructors? In Section 3.1, we used field types to encode information about a tagged value's presence or absence. Here, we need exactly the same information, but this time *about every tag*. So, we need to manipulate a family of field types, indexed by tags. To do so, we add one layer to the type algebra: *rows* of field types.

$$\begin{aligned} \tau &::= \alpha, \beta, \gamma, \ldots \mid \bot \mid \top \mid \tau \to \tau \mid [\,\rho\,] \\ \rho &::= \varphi, \psi, \ldots \mid K : \phi;\ \rho \mid \partial\phi \\ \phi &::= \varphi, \psi, \ldots \mid \texttt{Abs} \mid \texttt{Pre}\ \tau \mid \texttt{Any} \end{aligned}$$

We can now extend the previous section's proposal, as follows:

$$\begin{aligned} K &: \alpha \to [\,K : \texttt{Pre}\ \alpha;\ \partial\texttt{Abs}\,] \\ K^{-1} &: (\alpha \to \beta) \to ([\,K : \texttt{Abs};\ \psi\,] \to \gamma) \to [\,K : \varphi;\ \psi\,] \to \gamma \\ \text{where} \quad & \varphi \leq \texttt{Pre}\ \alpha \\ & \texttt{Pre} \leq \varphi\,?\,\beta \leq \gamma \\ \texttt{close} &: [\,\partial\texttt{Abs}\,] \to \bot \end{aligned}$$

K^{-1}'s type scheme involves the same constraints as in the basic case. Using a single row variable, namely ψ, in two distinct positions allows expressing the fact that values carrying any tag other than K shall be passed unmodified to K^{-1}'s second argument.

`close`'s argument type is $[\,\partial\texttt{Abs}\,]$, which prevents it from ever being invoked. This accords with the fact that `close` does not have an associated reduction rule. It plays the role of a function defined by zero cases.

This system offers *extensible* pattern matchings: k-ary *case* constructs may be written using k nested destructor applications and `close`, and receive the desired, accurate type. Thus, no specific language construct or type inference rule is needed to deal with them.

4 Record Concatenation

Static typing for record operations is a widely studied problem [4, 13]. Common operations include selection, extension, restriction, and concatenation. The latter comes in two flavors: symmetric and asymmetric. The former requires its arguments to have disjoint sets of fields, whereas the latter gives precedence to the second one when a conflict occurs.

Of these operations, concatenation is probably the most difficult to deal with, because its behavior varies according to the presence or absence of each field in its two arguments. This has led many authors to restrict their attention to type checking, and to not address the issue of type inference [7]. An inference algorithm for asymmetric concatenation was suggested by Wand [23]. He uses *disjunctions* of constraints, however, which gives his system exponential complexity. Rémy [19] suggests an encoding of concatenation into λ-abstraction and record extension, whence an inference algorithm may be derived. Unfortunately, its power is somewhat decreased by subtle interactions with ML's restricted polymorphism; furthermore, the encoding is exposed to the user. In later work [20], Rémy suggests a direct, constraint-based algorithm, which involves a special form of constraints. Our approach is inspired from this work, but re-formulated in terms of conditional constraints, thus showing that no ad hoc construct is necessary.

Again, our presentation is in two steps. The basic case, where records only have one field, is tackled using subtyping and conditional constraints. Then, rows allow us to easily transfer our results to the case of multiple fields.

4.1 The Basic Case

We assume a language equipped with one-field records, whose unique field may be either "absent" or "present". More precisely, we assume a constant data constructor `Abs`, and a unary data constructor `Pre`; a "record" is a value built with one of these constructors. A data destructor, $\texttt{Pre}^{-1}$, allows accessing the contents of a non-empty record. Lastly, the language offers asymmetric and symmetric concatenation primitives, written @ and @@, respectively.

$$e ::= \ldots \mid \texttt{Abs} \mid \texttt{Pre} \mid \texttt{Pre}^{-1} \mid @ \mid @@$$

The relationship between record creation and record access is expressed by a simple reduction rule:

$$\texttt{Pre}^{-1}\,(\texttt{Pre}\,v) \quad \text{reduces to} \quad v$$

The semantics of asymmetric record concatenation is given as follows:

$$\begin{array}{rcl} v_1 \,@\, \texttt{Abs} & \text{reduces to} & v_1 \\ v_1 \,@\, (\texttt{Pre}\,v_2) & \text{reduces to} & \texttt{Pre}\,v_2 \end{array}$$

(In each of these rules, the value v_1 is required to be a record.) Lastly, symmetric concatenation is defined by

$$\begin{array}{rcl} \texttt{Abs} \,@@\, v_2 & \text{reduces to} & v_2 \\ v_1 \,@@\, \texttt{Abs} & \text{reduces to} & v_1 \end{array}$$

(In these two rules, v_1 and v_2 are required to be records.)

The construction of our type algebra is similar to the one performed in Section 3.1. We introduce a (unary) record type constructor, as well as a distinction between regular types and field types:

$$\begin{array}{l} \tau ::= \alpha, \beta, \gamma, \ldots \mid \bot \mid \top \mid \tau \rightarrow \tau \mid \{\phi\} \\ \phi ::= \varphi, \psi, \ldots \mid \texttt{Bot} \mid \texttt{Abs} \mid \texttt{Pre}\ \tau \mid \texttt{Either}\ \tau \mid \texttt{Any} \end{array}$$

Let us explain, step by step, our definition of field types. Our first, natural step is to introduce type constructors Abs and Pre, which allow describing values built with the data constructors Abs and Pre. The former is a constant type constructor, while the latter is unary and covariant.

Many type systems for record languages define Pre τ to be a subtype of Abs. This allows a record whose field is present to pretend it is not, leading to a classic theory of records whose fields may be “forgotten” via subtyping. However, when the language offers record concatenation, such a definition isn’t appropriate. Why? Concatenation – asymmetric or symmetric – involves a choice between two reduction rules, which is performed by matching one, or both, of the arguments against the data constructors Abs and Pre. If, at the level of types, we allow a non-empty record to masquerade as an empty one, then it becomes impossible, based on the arguments’ types, to find out which rule applies, and to determine the type of the operation’s result. In summary, in the presence of record concatenation, no subtyping relationship must exist between Pre τ and Abs. (This problem is well described – although not solved – in [4].)

This leads us to making Abs and Pre *incomparable*. Once this choice has been made, completing the definition of field types is rather straightforward. Because our system requires type constructors to form a lattice, we define a least element

$\mathtt{Bot}$, and a greatest element $\mathtt{Any}$. Lastly, we introduce a unary, covariant type constructor, $\mathtt{Either}$, which we define as the least upper bound of $\mathtt{Abs}$ and $\mathtt{Pre}$, so that $\mathtt{Abs} \sqcup (\mathtt{Pre}\ \tau)$ equals $\mathtt{Either}\ \tau$. This optional refinement allows us to keep track of a field's type, even when its presence is not ascertained. The lattice of field types is shown in figure 1 on page 328.

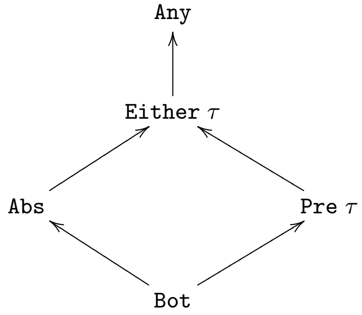


Fig. 1. The lattice of record field types

Let us now assign types to the primitive operations offered by the language. Record creation and access receive their usual types:

$$\begin{aligned} \mathtt{Abs} &: \{\mathtt{Abs}\} \\ \mathtt{Pre} &: \alpha \to \{\mathtt{Pre}\ \alpha\} \\ \mathtt{Pre}^{-1} &: \{\mathtt{Pre}\ \alpha\} \to \alpha \end{aligned}$$

There remains to come up with correct, precise types for both flavors of record concatenation. The key idea is simple. As shown by its operational semantics, (either flavor of) record concatenation is really a function defined by cases over the data constructors $\mathtt{Abs}$ and $\mathtt{Pre}$ – and Section 3 has shown how to accurately describe such a function. Let us begin, then, with asymmetric concatenation:

$$\begin{aligned} @ &: \{\varphi_1\} \to \{\varphi_2\} \to \{\varphi_3\} \\ \text{where} \quad & \varphi_2 \leq \mathtt{Either}\ \alpha_2 \\ & \mathtt{Abs} \leq \varphi_2\ ?\ \varphi_1 \leq \varphi_3 \\ & \mathtt{Pre} \leq \varphi_2\ ?\ \mathtt{Pre}\ \alpha_2 \leq \varphi_3 \end{aligned}$$

Clearly, each conditional constraint mirrors one of the reduction rules. In the second conditional constraint, we assume α_2 is the type of the second record's field – if it has one. The first subtyping constraint represents this assumption. Notice that we use $\mathtt{Pre}\ \alpha_2$, rather than φ_2, as the second branch's result type; this is strictly more precise, because φ_2 may be of the form $\mathtt{Either}\ \alpha_2$.

Lastly, we turn to symmetric concatenation:

$$\begin{aligned} @@ &: \{\varphi_1\} \rightarrow \{\varphi_2\} \rightarrow \{\varphi_3\} \\ \text{where} \quad & \texttt{Abs} \leq \varphi_1 \,?\, \varphi_2 \leq \varphi_3 \\ & \texttt{Abs} \leq \varphi_2 \,?\, \varphi_1 \leq \varphi_3 \\ & \texttt{Pre} \leq \varphi_1 \,?\, \varphi_2 \leq \texttt{Abs} \\ & \texttt{Pre} \leq \varphi_2 \,?\, \varphi_1 \leq \texttt{Abs} \end{aligned}$$

Again, each of the first two constraints mirrors a reduction rule. The last two constraints disallow the case where both arguments are non-empty records. (The careful reader will notice that any one of these two constraints would in fact suffice; both are kept for symmetry.)

In both cases, the operation's description in terms of constraints closely resembles its operational definition. Automatically deriving the former from the latter seems possible; this is an area for future research.

4.2 The General Case

We now move to a language with a denumerable set of record labels, written l, m, etc. The language allows creating the empty record, as well as any one-field record; it also offers selection and concatenation operations. Extension and restriction can be easily added, if desired; we shall dispense with them.

$$e ::= \varnothing \mid \{l = e\} \mid e.l \mid @ \mid @@$$

We do not give the language's semantics, which should hopefully be clear enough.

At the level of types, we again introduce rows of field types, denoted by ρ. Furthermore, we introduce rows of regular types, denoted by ϱ. Lastly, we lift the five field type constructors to the level of rows.

$$\begin{aligned} &\tau ::= \alpha, \beta, \gamma, \ldots \mid \bot \mid \top \mid \tau \rightarrow \tau \mid \{\rho\} \\ &\phi ::= \varphi, \psi, \ldots \mid \texttt{Bot} \mid \texttt{Abs} \mid \texttt{Pre}\ \tau \mid \texttt{Either}\ \tau \mid \texttt{Any} \\ &\varrho ::= \alpha, \beta, \gamma, \ldots \mid l : \tau;\ \varrho \mid \partial\tau \\ &\rho ::= \varphi, \psi, \ldots \mid l : \phi;\ \rho \mid \partial\phi \mid \texttt{Bot} \mid \texttt{Abs} \mid \texttt{Pre}\ \varrho \mid \texttt{Either}\ \varrho \mid \texttt{Any} \end{aligned}$$

This allows writing complex constraints between rows, such as $\varphi \leq \texttt{Pre}\ \alpha$, where φ and α are row variables. A constraint between rows stands for an infinite family of constraints between types, obtained component-wise. That is,

$$(l : \varphi';\ \varphi'') \leq \texttt{Pre}\ (l : \alpha';\ \alpha'') \quad \text{stands for} \quad (\varphi' \leq \texttt{Pre}\ \alpha') \wedge (\varphi'' \leq \texttt{Pre}\ \alpha'')$$

We may now give types to the primitive record operations. Creation and selection are easily dealt with:

$$\begin{aligned} \varnothing &: \{\partial\texttt{Abs}\} \\ \{l = \cdot\} &: \alpha \rightarrow \{l : \texttt{Pre}\ \alpha;\ \partial\texttt{Abs}\} \\ \cdot.l &: \{l : \texttt{Pre}\ \alpha;\ \partial\texttt{Any}\} \rightarrow \alpha \end{aligned}$$

Interestingly, the types of both concatenation operations are *unchanged* from the previous section – at least, syntactically. (For space reasons, we do not repeat them here.) A subtle difference lies in the fact that all variables involved must now be read as row variables, rather than as type variables. In short, the previous section exhibited constraints which describe concatenation, at the level of a single record field; here, the row machinery allows us to replicate these constraints over an infinite set of labels. This increase in power comes almost for free: it does not add any complexity to our notion of subtyping.

5 Dynamic Messages

So-called "dynamic" messages have recently received new attention in the static typing community. Bugliesi and Crafa [3] propose a higher-order type system which accounts for first-class messages. Nishimura [11] tackles the issue of type inference and suggests a second-order system à la Ohori [13]. Müller and Nishimura [10] propose a simplified approach, based on an extended feature logic.

The problem consists in performing type inference for an object-oriented language where messages are first-class values, made up of a *label* and a *parameter*. Here, we view objects as records of functions, and messages as tagged values. (Better ways of modeling objects exist, but that is an independent issue.) Thus, we consider a language with records and data constructors, as described in Sections 3.2 and 4.2. Furthermore, we let record labels and data constructors range over a single name space, that of message labels. (To save space, we choose to deal directly with the case of multiple message labels; however, our usual, two-step presentation would still be possible.) Lastly, we define a primitive message-send operation, written #, whose semantics is as follows:

$$\#\,\{m = v_1; \dots\}\,(m\,v_2) \quad \text{reduces to} \quad (v_1\,v_2)$$

In plain words, # examines its second argument, which must be some message m with parameter v_2. It then looks up the method named m in the receiver object, and applies the method's code, v_1, to the message parameter.

In a language with "static" messages, a message-send operation may only involve a constant message label. So, instead of a single message-send operation, a family thereof, indexed by message labels, is provided. In fact, in our simple model, these operations are definable within the language. The operation $\#m$, which allows sending the message m to some object o with parameter p, may be defined as $\lambda o.\lambda p.(o.m\ p)$. Then, type inference yields

$$\#m : \{m : \mathtt{Pre}\ (\alpha \rightarrow \beta);\ \partial\mathtt{Any}\} \rightarrow \alpha \rightarrow \beta$$

Because the message label, m, is statically known, it may be explicitly mentioned in the type scheme, making it easy to require the receiver object to carry an appropriate method. In a language with "dynamic" messages, on the other hand, m is no longer known. The problem thus appears more complex; it has, in fact, sparked the development of special-purpose constraint languages [10]. Yet, the machinery introduced so far in this paper suffices to solve it.

Consider the partial application of the message send primitive # to some record r. It is a function which accepts some tagged value $(m\,v)$, then invokes an appropriate piece of code, selected according to the label m. This should ring a bell – it is merely a form of pattern matching, which this paper has extensively discussed already. Therefore, we propose

$$\begin{array}{rl} & \# : \{\varphi\} \rightarrow [\,\psi\,] \rightarrow \beta \\ \text{where} & \psi \leq \texttt{Pre}\ \alpha \\ & \texttt{Pre} \leq \psi \,?\, \varphi \leq \texttt{Pre}\ (\alpha \rightarrow \partial\beta) \end{array}$$

(Here, all variables except β are row variables.) The operation's first (resp. second) argument is required to be an object (resp. a message), whose contents (resp. possible values) are described by the row variable φ (resp. ψ). The first constraint merely lets α stand for the message parameter's type. The conditional constraint, which involves two row terms, should again be understood as a family, indexed by message labels, of conditional constraints between record field types. The conditional constraint associated with some label m shall be triggered only if ψ's element at index m is of the form $\texttt{Pre}\ \_$, i.e. only if the message's label may be m. When it is triggered, its right-hand side becomes active, with a three-fold effect. First, φ's element at index m must be of the form $\texttt{Pre}\ (\_ \rightarrow \_)$, i.e. the receiver object must carry a method labeled m. Second, the method's argument type must be (a supertype of) α's element at label m, i.e. the method must be able to accept the message's parameter. Third, the method's result type must be (a subtype of) β, i.e. the whole operation's result type must be (at least) the join of all potentially invoked methods' return types.

Our proposal shows that type inference for "dynamic" messages requires no dedicated theoretical machinery. It also shows that "dynamic" messages are naturally compatible with all operations on records, including concatenation – a question which was left unanswered by Nishimura [11].

6 Conclusion

In this paper, we have advocated enriching an existing constraint-based type inference framework [15] with rows [18] and conditional constraints [2]. This provides a single (and simple) solution to several difficult type inference problems, each of which seemed to require, until now, special forms of constraints. From a practical point of view, it allows them to benefit from known constraint simplification techniques [17], leading to an efficient inference algorithm [14].

We believe our system subsumes Rémy's proposal for record concatenation [20], as well as Müller and Nishimura's view of "dynamic" messages [10]. Aiken, Wimmers and Lakshman's "soft" type system [2] is more precise than ours, because it interprets constraints in a richer logical model, but otherwise offers similar features. In fact, the ideas developed in this paper could have been presented in the setting of Bane [5], or, more generally, of any system which allows writing sufficiently expressive constrained type schemes.

References

[1] Alexander S. Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Functional Programming & Computer Architecture*, pages 31–41. ACM Press, June 1993. URL: http://http.cs.berkeley.edu/~aiken/ftp/fpca93.ps.

[2] Alexander S. Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Principles of Programming Languages*, pages 163–173, January 1994. URL: http://http.cs.berkeley.edu/~aiken/ftp/popl94.ps.

[3] Michele Bugliesi and Silvia Crafa. Object calculi for dynamic messages. In *The Sixth International Workshop on Foundations of Object-Oriented Languages, FOOL 6, San Antonio, Texas*, January 1999. URL: ftp://ftp.cs.williams.edu/pub/kim/FOOL6/bugliesi.ps.

[4] Luca Cardelli and John Mitchell. Operations on records. *Mathematical Structures in Computer Science*, 1:3–48, 1991. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, MIT Press, 1994. URL: http://research.microsoft.com/Users/luca/Papers/Records.ps.

[5] Manuel Fähndrich. Bane*: A Library for Scalable Constraint-Based Program Analysis*. PhD thesis, University of California at Berkeley, 1999. URL: http://research.microsoft.com/~maf/diss.ps.

[6] Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 235–248, Las Vegas, Nevada, June 1997. URL: http://www.cs.rice.edu/CS/PLT/Publications/pldi97-ff.ps.gz.

[7] Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages (POPL '91)*, pages 131–142, Orlando, Florida, January 1991. ACM Press. URL: http://www.cis.upenn.edu/~bcpierce/papers/merge.ps.gz.

[8] Nevin Heintze. Set based analysis of ML programs. Technical Report CMU-CS-93-193, Carnegie Mellon University, School of Computer Science, July 1993. URL: ftp://reports.adm.cs.cmu.edu/usr/anon/1993/CMU-CS-93-193.ps.

[9] Martin Müller, Joachim Niehren, and Andreas Podelski. Ordering constraints over feature trees. *Constraints, an International Journal, Special Issue on CP'97, Linz, Austria*, 1999. URL: ftp://ftp.ps.uni-sb.de/pub/papers/ProgrammingSysLab/ftsub-constraints-99.ps.gz.

[10] Martin Müller and Susumu Nishimura. Type inference for first-class messages with feature constraints. In Jieh Hsiang and Atsushi Ohori, editors, *Asian Computer Science Conference (ASIAN 98)*, volume 1538 of *LNCS*, pages 169–187, Manila, The Philippines, December 1998. Springer-Verlag. URL: ftp://ftp.ps.uni-sb.de/pub/papers/ProgrammingSysLab/FirstClass98.ps.gz.

[11] Susumu Nishimura. Static typing for dynamic messages. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 266–278, San Diego, California, January 1998. URL: ftp://ftp.kurims.kyoto-u.ac.jp/pub/paper/member/nisimura/dmesg-popl98.ps.gz.

[12] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1), 1999. URL: http://www.cs.yale.edu/~sulzmann-martin/publications/tapos.ps.

[13] Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, November 1995.

[14] François Pottier. `Wallace`: an efficient implementation of type inference with subtyping. URL: `http://pauillac.inria.fr/~fpottier/wallace/`.

[15] François Pottier. Simplifying subtyping constraints: a theory. Submitted for publication, December 1998. URL: `http://pauillac.inria.fr/~fpottier/publis/fpottier-journal-98.ps.gz`.

[16] François Pottier. Type inference in the presence of subtyping: from theory to practice. Technical Report 3483, INRIA, September 1998. URL: `ftp://ftp.inria.fr/INRIA/publication/RR/RR-3483.ps.gz`.

[17] François Pottier. Subtyping-constraint-based type inference with conditional constraints: algorithms and proofs. Unpublished draft, July 1999. URL: `http://pauillac.inria.fr/~fpottier/publis/fpottier-conditional.ps.gz`.

[18] Didier Rémy. Projective ML. In *1992 ACM Conference on Lisp and Functional Programming*, pages 66–75, New-York, 1992. ACM Press. URL: `ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/lfp92.ps.gz`.

[19] Didier Rémy. Typing record concatenation for free. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1993. URL: `ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/taoop2.ps.gz`.

[20] Didier Rémy. A case study of typechecking with constrained types: Typing record concatenation. Presented at the workshop on Advances in Types for Computer Science at the Newton Institute, Cambridge, UK, August 1995. URL: `http://cristal.inria.fr/~remy/work/sub-concat.dvi.gz`.

[21] John C. Reynolds. Automatic computation of data set definitions. In A. J. H. Morrell, editor, *Information Processing 68*, volume 1, pages 456–461, Amsterdam, 1969. North-Holland.

[22] Valery Trifonov and Scott Smith. Subtyping constrained types. In *Proceedings of the Third International Static Analysis Symposium*, volume 1145 of *LNCS*, pages 349–365. SV, September 1996. URL: `http://www.cs.jhu.edu/~trifonov/papers/subcon.ps.gz`.

[23] Mitchell Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, pages 1–15, 1993. A preliminary version appeared in *Proc. 4th IEEE Symposium on Logic in Computer Science* (1989), pp. 92–97. URL: `ftp://ftp.ccs.neu.edu/pub/people/wand/papers/ic-91.dvi`.

A Rules

This appendix gives a short description of the system's type inference rules (Figure 2). Even though only the core language is explicitly treated, these rules are sufficient to deal with a full-featured programming language. Indeed, any extra language construct may be viewed either as syntactic sugar, or as a new primitive operation, which can be bound in an initial typing environment Γ_0. Also, note that these type inference rules use neither conditional constraints, nor rows; these will come only from Γ_0.

For simplicity, we distinguish identifiers bound by λ, denoted $x, y, \ldots$ from those bound by `let`, denoted $X, Y, \ldots$ Furthermore, we expect λ-identifiers to be unique; that is, each λ-identifier must be bound at most once in a given

$$\frac{\alpha \text{ fresh}}{\Gamma \vdash_{\mathrm{I}} x : \forall\varnothing.\, \langle x : \alpha\rangle \Rightarrow \alpha} \qquad (\textsc{Var}_{\mathrm{I}})$$

$$\frac{\Gamma \vdash_{\mathrm{I}} e : \forall C.\, A \Rightarrow \tau' \qquad A(x) = \tau}{\Gamma \vdash_{\mathrm{I}} \lambda x.e : \forall C.\, (A \setminus x) \Rightarrow \tau \to \tau'} \qquad (\textsc{Abs}_{\mathrm{I}})$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{I}} e_1 : \forall C_1.\, A_1 \Rightarrow \tau_1 \qquad \Gamma \vdash_{\mathrm{I}} e_2 : \forall C_2.\, A_2 \Rightarrow \tau_2 \\ \alpha \text{ fresh} \qquad C = C_1 \cup C_2 \cup \{\tau_1 \leq \tau_2 \to \alpha\}\end{array}}{\Gamma \vdash_{\mathrm{I}} e_1\, e_2 : \forall C.\, (A_1 \sqcap A_2) \Rightarrow \alpha} \qquad (\textsc{App}_{\mathrm{I}})$$

$$\frac{\Gamma(X) = \sigma \qquad \rho \text{ fresh renaming of } \sigma}{\Gamma \vdash_{\mathrm{I}} X : \rho(\sigma)} \qquad (\textsc{LetVar}_{\mathrm{I}})$$

$$\frac{\Gamma \vdash_{\mathrm{I}} e_1 : \sigma_1 \qquad \Gamma + [X \mapsto \sigma_1] \vdash_{\mathrm{I}} e_2 : \sigma_2}{\Gamma \vdash_{\mathrm{I}} \texttt{let } X = e_1 \texttt{ in } e_2 : \sigma_2} \qquad (\textsc{Let}_{\mathrm{I}})$$

Fig. 2. Type inference rules

program. Lastly, in every expression of the form `let` $X = e_1$ `in` e_2, we require X to appear free within e_2. It would be easy to overcome these restrictions, at the expense of heavier notation.

The rules are fairly straightforward. The main point of interest is the way each application node produces a subtyping constraint. The only peculiarity is in the way type environments are dealt with. The *environment* Γ, which appears on the left of the turnstile, is a list of bindings of the form $X : \sigma$. Type schemes are slightly more complex than initially shown in Section 2. They are, in fact, of the form $\sigma ::= \forall C.\, A \Rightarrow \tau$, where the *context* A is a set of bindings of the form $x : \tau$. The point of such a formulation is to obtain a system where no type scheme has free type variables. This allows a simpler theoretical description of constraint simplification.

As far as notation is concerned, $\langle x : \alpha\rangle$ represents a context consisting of a single entry, which binds x to α. $A \setminus x$ is the context obtained by removing x's binding from A, if it exists. For the sake of readability, we have abused notation slightly. In rule ($\textsc{Abs}_{\mathrm{I}}$), $A(x)$ stands for the type associated with x in A, if A contains a binding for x; it stands for $\top$ otherwise. In rule ($\textsc{App}_{\mathrm{I}}$), $A_1 \sqcap A_2$ represents the point-wise intersection of A_1 and A_2. That is, whenever x has a binding in A_1 or A_2, its binding in $A_1 \sqcap A_2$ is $A_1(x) \sqcap A_2(x)$. Because we do not have intersection types, this expression should in fact be understood as a fresh type variable, accompanied by an appropriate conjunction of subtyping constraints.

The rules implicitly require every constraint set to admit at least one solution. Constraint solving and simplification are described in [15, 17].

B Examples

Example 1. We define a function which reads field l out of a record r, returning a default value d if r has no such field, by setting $\mathtt{extract} = \lambda d.\lambda r.(\{l = d\} \mathbin{@} r).l$. In our system, `extract`'s inferred type is

$$\begin{array}{rlll}\mathtt{extract} : & \alpha \rightarrow \{l : \varphi;\ \psi\} \rightarrow \gamma & & \\ \text{where} & \varphi \leq \mathtt{Either}\,\beta & & \psi \leq \mathtt{Either}\,\epsilon \\ & \mathtt{Abs} \leq \varphi\,?\,\alpha \leq \gamma & & \mathtt{Abs} \leq \psi\,?\,\mathtt{Abs} \leq \mathtt{Any} \\ & \mathtt{Pre} \leq \varphi\,?\,\beta \leq \gamma & & \mathtt{Pre} \leq \psi\,?\,\mathtt{Pre}\,\epsilon \leq \mathtt{Any}\end{array}$$

The first constraint retrieves $r.l$'s type and names it β, regardless of the field's presence. (If the field turns out to be absent, β will be unconstrained.) The left-hand conditional constraints clearly specify the dependency between the field's presence and the function's result.

The right-hand conditional constraints have tautologous conclusions – therefore, they are superfluous. They remain only because our current constraint simplification algorithms are "lazy" and ignore any conditional constraints whose condition has not yet been fulfilled. This problem could be fixed by implementing slightly more aggressive simplification algorithms.

The type inferred for $\mathtt{extract}\ 0\ \{l = 1\}$ and $\mathtt{extract}\ 0\ \{m = 1\}$ is `int`. Thus, in many cases, one need not be aware of the complexity hidden in `extract`'s type.

Example 2. We assume given an object o, of the following type:

$$\begin{array}{ll} o : \{ & \mathtt{getText} : \mathtt{Pre}\ (\mathtt{unit} \rightarrow \mathtt{string});\quad \mathtt{setText} : \mathtt{Pre}\ (\mathtt{string} \rightarrow \mathtt{unit}); \\ & \mathtt{select} : \mathtt{Pre}\ (\mathtt{int} \times \mathtt{int} \rightarrow \mathtt{unit});\ \partial\mathtt{Abs}\quad \}\end{array}$$

o may represent, for instance, an editable text field in a graphic user interface system. Its methods allow programmatically getting and setting its contents, as well as selecting a portion of text.

Next, we assume a list data structure, equipped with a simple iterator:

$$\mathit{iter} : (\alpha \rightarrow \mathtt{unit}) \rightarrow \alpha\ \mathtt{list} \rightarrow \mathtt{unit}$$

The following expression creates a list of messages, and uses *iter* to send each of them in turn to o:

$$\mathit{iter}\,(\#\,o)\,[\,\mathtt{setText}\ \text{“Hello!”};\ \mathtt{select}\,(0, 5)\,]$$

This expression is well-typed, because o contains appropriate methods to deal with each of these messages, and because these methods return `unit`, as expected by *iter*. The expression's type is of course `unit`, *iter*'s return type.

Here is a similar expression, which involves a `getText` message:

$$\mathit{iter}\,(\#\,o)\,[\,\mathtt{setText}\ \text{“Hello!”};\ \mathtt{getText}\,()\,]$$

This time, it is ill-typed. Indeed, sending a `setText` message to o produces a result of type `unit`, while sending it a `getText` message produces a result of type `string`. Thus, $(\#\,o)$'s result type must be $\top$, the join of these types. This makes $(\#\,o)$ an unacceptable argument for *iter*, since the latter expects a function whose return type is `unit`.

First-Class Structures for Standard ML

Claudio V. Russo*

Cambridge University Computer Laboratory, Cambridge CB2 3QG, UK
cvr21@cl.cam.ac.uk

Abstract. Standard ML is a statically typed programming language that is suited for the construction of both small and large programs. "Programming in the small" is captured by Standard ML's *Core* language. "Programming in the large" is captured by Standard ML's *Modules* language that provides constructs for organising related Core language definitions into self-contained modules with descriptive interfaces. While the Core is used to express details of algorithms and data structures, Modules is used to express the overall *architecture* of a software system. The Modules and Core languages are *stratified* in the sense that modules may not be manipulated as ordinary values of the Core. This is a limitation, since it means that the architecture of a program cannot be reconfigured according to run-time demands. We propose a novel extension of the language that allows modules to be manipulated as first-class values of the Core language. The extension greatly extends the expressive power of the language and has been shown to be compatible with both Core type inference and a separate extension to higher-order modules.

1 Introduction

Standard ML [10] is a high-level programming language that is suited for the construction of both small and large programs.

Standard ML's general-purpose *Core* language supports "programming in the small" with a rich range of types and computational constructs that includes recursive types and functions, control constructs, exceptions and references.

Standard ML's special-purpose *Modules* language supports "programming in the large". Constructed on top of the Core, the Modules language allows definitions of identifiers denoting Core language types and terms to be packaged together into possibly nested *structures*, whose components are accessed by the dot notation. Structures are *transparent*: by default, the *realisation* (i.e. implementation) of a type component within a structure is evident outside the structure. *Signatures* are used to specify the types of structures, by specifying their individual components. A type component may be specified *opaquely*, permitting a variety of realisations, or *transparently*, by equating it with a particular Core type. A structure *matches* a signature if it provides an implementation for

* This research was completed at the LFCS, Division of Informatics, University of Edinburgh under EPSRC grant GR/K63795. Thanks to Don Sannella, Healfdene Goguen and the anonymous referees.

G. Smolka (Ed.): ESOP/ETAPS 2000, LNCS 1782, pp. 336–350, 2000.

all of the specified components, and, thanks to *subtyping*, possibly more. A signature may be used to *opaquely constrain* a matching structure. This existentially quantifies over the actual realisation of type components that have opaque specifications in the signature, effectively hiding their implementation. A *functor* definition defines a polymorphic function mapping structures to structures. A functor may be *applied* to any structure that realises a subtype of the formal argument's type, resulting in a concrete implementation of the functor body.

Despite the flexibility of the Modules type system, the notion of computation at the Modules level is actually very weak, permitting only functor application, to model the linking of structures, and projection, to provide access to a structure's components. Moreover, the stratification between Core and Modules means that the stronger computational mechanisms of the Core cannot be exploited in the construction of structures. This severe limitation means that the architecture of a program cannot be reconfigured according to run-time demands. For instance, we cannot dynamically choose between the various back-ends of a cross compiler, if those back-ends are implemented as separate structures.

In this paper, we relax the Core/Modules stratification, allowing structures to be manipulated as *first-class* citizens of the Core language. Our extension allows structures to be passed as arguments to Core functions, returned as results of Core computations, stored in Core data structures and so on.

For presentation purposes, we formulate our extension for a representative toy language called Mini-SML. The static semantics of Mini-SML is based directly on that of Standard ML. Mini-SML includes the essential features of Standard ML Modules but, for brevity, only has a simple Core language of explicitly typed, monomorphic functions ([16] treats a Standard ML-like Core). Section 2 introduces the syntax of Mini-SML. Section 3 gives a motivating example to illustrate the limitations of the Core/Modules stratification. Section 4 reviews the static semantics of Mini-SML. Section 5 defines our extension to first-class structures. Section 6 revisits the motivating example to show the utility of our extension. Section 7 presents a different example to demonstrate that Mini-SML becomes more expressive with our extension. Section 8 discusses our contribution.

2 The Syntax of Mini-SML

The *type* and *term* syntax of Mini-SML is defined by the grammar in Figures 1 and 2, where $\mathrm{t} \in \mathrm{TypId}$, $\mathrm{x} \in \mathrm{ValId}$, $\mathrm{X} \in \mathrm{StrId}$, $\mathrm{F} \in \mathrm{FunId}$ and $\mathrm{T} \in \mathrm{SigId}$ range over disjoint sets of type, value, structure, functor and signature identifiers.

A *core type* u may be used to define a type identifier or to specify the type of a Core value. These are just the types of a simple functional language, extended with the projection sp.t of a type component from a structure path. A *signature body* B is a sequential specification of a structure's components. A type component may be specified *transparently*, by equating it with a type, or *opaquely*, permitting a variety of realisations. Transparent specifications may be used to express *type sharing* constraints in the usual way. Value and structure components are specified by their type and signature. The specifications in a body

Core Types	u ::= t	type identifier
	$\mid$ u → u′ $\mid$ **int**	function space, integers
	$\mid$ sp.t	type projection
Signature Bodies	B ::= **type** t = u; B	transparent type specification
	$\mid$ **type** t; B	opaque type specification
	$\mid$ **val** x : u; B	value specification
	$\mid$ **structure** X : S; B	structure specification
	$\mid$ ϵ_B	empty body
Signature Expressions	S ::= **sig** B **end**	encapsulated body
	$\mid$ T	signature identifier

Fig. 1. Type Syntax of Mini-SML

Core Expressions	e ::= x	value identifier
	$\mid$ λx : u.e $\mid$ e e′	function, application
	$\mid$ *i* $\mid$ **ifzero** e **then** e′**else** e″	integer, zero test
	$\mid$ **fix** e	fixpoint of e (recursion)
	$\mid$ sp.x	value projection
Structure Paths	sp ::= X	structure identifier
	$\mid$ sp.X	structure projection
Structure Bodies	b ::= **type** t = u; b	type definition
	$\mid$ **val** x = e; b	value definition
	$\mid$ **structure** X = s;b	structure definition
	$\mid$ **functor** F (X : S) = s; b	functor definition
	$\mid$ **signature** T = S; b	signature definition
	$\mid$ ϵ_b	empty body
Structure Expressions	s ::= sp	structure path
	$\mid$ **struct** b **end**	structure body
	$\mid$ F(s)	functor application
	$\mid$ s :> S	opaque constraint

Fig. 2. Term Syntax of Mini-SML

are dependent in that subsequent specifications may refer to previous ones. A *signature expression* S encapsulates a body, or is a reference to a bound signature identifier. A structure *matches* a signature expression if it provides an implementation for all of the specified components, and possibly more.

Core expressions e describe a simple functional language extended with the projection of a value identifier from a structure path. A *structure path* sp is a reference to a bound structure identifier, or the projection of one of its substructures. A *structure body* b is a dependent sequence of definitions: subsequent definitions may refer to previous ones. A type definition abbreviates a type. Value and structure definitions bind term identifiers to the values of expressions. A functor definition introduces a named function on structures: X is the functor's formal argument, S specifies the argument's type, and s is the functor's body that may refer to X. The functor may be applied to any argument that matches S. A signature definition abbreviates a signature. A *structure expression* s evaluates to a structure. It may be a path or an encapsulated structure body, whose

```
signature Stream = sig type nat = int; type state;
                       val start: state;
                       val next: state → state;
                       val value: state → nat
                   end;
structure TwoOnwards = struct type nat = int; type state = nat;
                              val start = 2;
                              val next = λs:state.succ s;
                              val value = λs:state.s
                       end;
signature State = Stream;
structure Start = TwoOnwards:>State;
functor Next (S:State) =
       struct type nat = S.nat; type state = S.state;
              val filter = fix λfilter:state→state.
                      λs:state. ifzero mod (S.value s) (S.value S.start)
                                then filter (S.next s) else s;
              val start = filter S.start;
              val next = λs:state.filter (S.next s);
              val value = S.value
       end;
functor Value (S:State) = struct val value = S.value (S.start) end
```

Fig. 3. Using structures to implement streams and a stratified, but useless, Sieve.

type, value and structure definitions (but not functor or signature definitions) become the components of the structure. The application of a functor evaluates its body with respect to the value of the actual argument. An opaque constraint restricts the visibility of the structure's components to those specified in the signature, which the structure must match, and hides the actual realisations of type components with opaque specifications, introducing new abstract types.

By supporting local functor and signature definitions, structure bodies can play the role of Standard ML's separate top-level syntax. [18] formalises recursive datatypes, local structure definitions and transparent signature constraints.

3 Motivating Example: The Sieve of Eratosthenes

We can illustrate the limitations of the Core/Modules stratification of Mini-SML (and Standard ML) by attempting to implement the Sieve of Eratosthenes using Modules level structures as the fundamental "data structure". It is a moot point that the Sieve can be coded directly in the Core: our aim is to highlight the shortcomings of second-class modules. The example is adapted from [12].

The Sieve is a well-known algorithm for calculating the infinite list, or *stream*, of prime (natural) numbers. We can represent such a stream as a "process", defined by a specific representation *nat* of the set of natural numbers, an unspecified set *state* of internal states, a designated initial or *start* state, a transition function taking us from one state to the *next* state, and a function *value* returning the natural number associated with each state. Reading the values off the process's sequence of states yields the stream.

Given a stream s, let $\mathit{sift}(s)$ be the substream of s consisting of those values not divisible by the initial value of s. Viewed as a process, the states of $\mathit{sift}(s)$ are just the states of s, filtered by the removal of any states whose values are divisible by the value of s's start state. The stream of primes is obtained by taking the initial value of each stream in the sequence of streams:

$$\begin{array}{lcccccccccccccl}
\mathit{twoonwards} & = & \mathbf{2}, & 3, & 4, & 5, & 6, & 7, & 8, & 9, & 10, & 11, & 12, & \ldots \\
\mathit{sift}(\mathit{twoonwards}) & = & & \mathbf{3}, & & 5, & & 7, & & 9, & & 11, & & \ldots \\
\mathit{sift}(\mathit{sift}(\mathit{twoonwards})) & = & & & & \mathbf{5}, & & 7, & & & & 11, & & \ldots \\
 & \vdots & & & & & & & & & & & &
\end{array}$$

The Sieve of Eratosthenes represents this construction as the following process. The *states* of the Sieve are streams. The Sieve's *start* state is the stream *twoonwards*. The *next* state of the Sieve is obtained by *sift*ing the current state. The *value* of each state of the Sieve is the first value of that state viewed as a stream. Observe that our description of the Sieve also describes a stream.

Consider the code in Fig. 3. Given our description of streams as processes, it seems natural to use structures matching the signature `Stream` to implement streams: e.g. the structure `TwoOnwards` implements the stream *twoonwards*. The remaining code constructs an implementation of the Sieve. The states of the Sieve are structures matching the signature `State` (i.e. `Stream`). The `Start` state of the Sieve is the structure `TwoOnwards`. The functor `Next` takes a structure `S` matching `State` and returns a sifted structure that also matches `State`. The functor `Value` returns the value of a state of the Sieve, by returning the initial value of the state viewed as a stream.

Now we can indeed calculate the value of the n^{th} prime (counting from 0):

```
structure NthValue = Value(Next(···Next(Start)···));
val nthprime = NthValue.value
```

by chaining n applications (underlined above) of the functor `Next` to `Start` and then extracting the resulting value. The problem is that we can only do this for a *fixed* n: because of the stratification of Core and Modules, it is impossible to implement the mathematical function that returns the nth state of the Sieve for an *arbitrary* n. It cannot be implemented as a Core function, even though the Core supports iteration, because the states of the Sieve are structures that do not belong to the Core language. It cannot be implemented as a Modules functor, because the computation on structures is limited to functor application and projection, which is too weak to express iteration. This means that our implementation of the Sieve is useless.

Notice also that, in this implementation, the components of the Sieve do not describe a stream in the sense of the signature `Stream`: the states of the Sieve are structures, not values of the Core and the state transition function is a functor, not a Core function. Our implementation fails to capture the impredicative description of the Sieve as a stream constructed from streams.

$$
\begin{array}{rcll}
\alpha \in \mathit{Var} & \overset{\text{def}}{=} & \{\alpha, \beta, \delta, \gamma, \ldots\} & \text{type variables} \\
P, Q \in \mathit{VarSet} & \overset{\text{def}}{=} & \mathrm{Fin}(\mathit{Var}) & \text{sets of type variables} \\
u \in \mathit{Type} & ::= & \alpha \;\mid\; u \to u' \;\mid\; \mathrm{int} & \text{type variable, function space, integers} \\
\varphi \in \mathit{Real} & \overset{\text{def}}{=} & \mathit{Var} \overset{\text{fin}}{\to} \mathit{Type} & \text{realisations} \\
\mathcal{S} \in \mathit{Str} & \overset{\text{def}}{=} & \left\{ \begin{array}{l|l} \mathcal{S}_{\mathrm{t}} \cup & \mathcal{S}_{\mathrm{t}} \in \mathrm{TypId} \overset{\text{fin}}{\to} \mathit{Type}, \\ \mathcal{S}_{\mathrm{x}} \cup & \mathcal{S}_{\mathrm{x}} \in \mathrm{ValId} \overset{\text{fin}}{\to} \mathit{Type}, \\ \mathcal{S}_{\mathrm{X}} & \mathcal{S}_{\mathrm{X}} \in \mathrm{StrId} \overset{\text{fin}}{\to} \mathit{Str} \end{array} \right\} & \text{semantic structures} \\
\mathcal{L} \in \mathit{Sig} & ::= & \Lambda P.\mathcal{S} & \text{semantic signatures} \\
\mathcal{X} \in \mathit{ExStr} & ::= & \exists P.\mathcal{S} & \text{existential structures} \\
\mathcal{F} \in \mathit{Fun} & ::= & \forall P.\mathcal{S} \to \mathcal{X} & \text{semantic functors} \\
\mathcal{C} \in \mathit{Context} & \overset{\text{def}}{=} & \left\{ \begin{array}{l|l} \mathcal{C}_{\mathrm{t}} \cup & \mathcal{C}_{\mathrm{t}} \in \mathrm{TypId} \overset{\text{fin}}{\to} \mathit{Type}, \\ \mathcal{C}_{\mathrm{T}} \cup & \mathcal{C}_{\mathrm{T}} \in \mathrm{SigId} \overset{\text{fin}}{\to} \mathit{Sig}, \\ \mathcal{C}_{\mathrm{x}} \cup & \mathcal{C}_{\mathrm{x}} \in \mathrm{ValId} \overset{\text{fin}}{\to} \mathit{Type}, \\ \mathcal{C}_{\mathrm{X}} \cup & \mathcal{C}_{\mathrm{X}} \in \mathrm{StrId} \overset{\text{fin}}{\to} \mathit{Str}, \\ \mathcal{C}_{\mathrm{F}} & \mathcal{C}_{\mathrm{F}} \in \mathrm{FunId} \overset{\text{fin}}{\to} \mathit{Fun} \end{array} \right\} & \text{semantic contexts}
\end{array}
$$

Notation. For sets A and B, $\mathrm{Fin}(A)$ denotes the set of *finite subsets* of A, and $A \overset{\text{fin}}{\to} B$ denotes the set of *finite maps* from A to B. Let f and g be finite maps. $\mathcal{D}(f)$ denotes the *domain of definition* of f. The finite map $f + g$ has domain $\mathcal{D}(f) \cup \mathcal{D}(g)$ and values $(f + g)(a) \overset{\text{def}}{=}$ if $a \in \mathcal{D}(g)$ then $g(a)$ else $f(a)$.

Fig. 4. Semantic Objects of Mini-SML

Once we allow structures as first-class citizens of the Core language, these problems disappear.

4 Review: The Static Semantics of Mini-SML

Before we can propose our extension, we need to present the static semantics, or typing judgements, of Mini-SML. Following Standard ML [10], the static semantics of Mini-SML distinguishes syntactic types of the language from their semantic counterparts, called *semantic objects*. Semantic objects play the role of types in the semantics. Figure 4 defines the semantic objects of Mini-SML. We let $\mathcal{O}$ range over all semantic objects.

Type variables $\alpha \in \mathit{Var}$ are just variables ranging over *semantic types* $u \in \mathit{Type}$. The latter are the semantic counterparts of syntactic Core types, and are used to record the denotations of type identifiers and the types of value identifiers. The symbols Λ, $\exists$ and $\forall$ are used to bind finite sets of type variables.

A *realisation* $\varphi \in \mathit{Real}$ maps type variables to semantic types and defines a *substitution* on type variables in the usual way. The operation of applying a realisation φ to an object $\mathcal{O}$ is written $\varphi(\mathcal{O})$.

Semantic structures $\mathcal{S} \in \mathit{Str}$ are used as the types of structure identifiers and paths. A semantic structure maps type components to the types they denote, and value and structure components to the types they inhabit. For clarity, we define the extension functions $\mathrm{t} \triangleright u, \mathcal{S} \overset{\mathrm{def}}{=} \{\mathrm{t} \mapsto u\} + \mathcal{S}$, $\mathrm{x} : u, \mathcal{S} \overset{\mathrm{def}}{=} \{\mathrm{x} \mapsto u\} + \mathcal{S}$, and $\mathrm{X} : \mathcal{S}, \mathcal{S}' \overset{\mathrm{def}}{=} \{\mathrm{X} \mapsto \mathcal{S}\} + \mathcal{S}'$, and let $\epsilon_{\mathcal{S}}$ denote the empty structure $\emptyset$.

A *semantic signature* $\Lambda P.\mathcal{S}$ is a parameterised type: it describes the family of structures $\varphi(\mathcal{S})$, for φ a realisation of the parameters in P.

The *existential structure* $\exists P.\mathcal{S}$, on the other hand, is a quantified type: variables in P are existentially quantified in $\mathcal{S}$ and thus abstract. Existential structures describe the types of structure bodies and expression. Existentially quantified type variables are explicitly introduced by opaque constraints $\mathrm{s} :> \mathrm{S}$, and implicitly eliminated at various points in the static semantics.

A *semantic functor* $\forall P.\mathcal{S} \to \mathcal{X}$ describes the type of a functor identifier: the universally quantified variables in P are bound simultaneously in the functor's domain, $\mathcal{S}$, and its range, $\mathcal{X}$. These variables capture the type components of the domain on which the functor behaves polymorphically; their possible occurrence in the range caters for the propagation of type identities from the functor's actual argument: functors are polymorphic functions on structures.

A *context* $\mathcal{C}$ maps type and signature identifiers to the types and signatures they denote, and maps value, structure and functor identifiers to the types they inhabit. For clarity, we define the extension functions $\mathcal{C}, \mathrm{t} \triangleright u \overset{\mathrm{def}}{=} \mathcal{C} + \{\mathrm{t} \mapsto u\}$, $\mathcal{C}, \mathrm{T} \triangleright \mathcal{L} \overset{\mathrm{def}}{=} \mathcal{C} + \{\mathrm{T} \mapsto \mathcal{L}\}$, $\mathcal{C}, \mathrm{x} : u \overset{\mathrm{def}}{=} \mathcal{C} + \{\mathrm{x} \mapsto u\}$, $\mathcal{C}, \mathrm{X} : \mathcal{S} \overset{\mathrm{def}}{=} \mathcal{C} + \{\mathrm{x} \mapsto \mathcal{S}\}$, and $\mathcal{C}, \mathrm{F} : \mathcal{F} \overset{\mathrm{def}}{=} \mathcal{C} + \{\mathrm{F} \mapsto \mathcal{F}\}$.

We let $\mathcal{V}(\mathcal{O})$ denote the set of variables occurring *free* in $\mathcal{O}$, where the notions of free and bound variable are defined as usual. Furthermore, we *identify* semantic objects that differ only in a renaming of bound type variables (α-conversion).

The operation of applying a realisation to a type (substitution) is extended to all semantic objects in the usual, capture-avoiding way.

Definition 1 (Enrichment Relation) *Given two structures $\mathcal{S}$ and $\mathcal{S}'$, $\mathcal{S}$* enriches *$\mathcal{S}'$, written $\mathcal{S} \succeq \mathcal{S}'$, if and only if $\mathcal{D}(\mathcal{S}) \supseteq \mathcal{D}(\mathcal{S}')$ and*

- *for all* $\mathrm{t} \in \mathcal{D}(\mathcal{S}')$, $\mathcal{S}(\mathrm{t}) = \mathcal{S}'(\mathrm{t})$,
- *for all* $\mathrm{x} \in \mathcal{D}(\mathcal{S}')$, $\mathcal{S}(\mathrm{x}) = \mathcal{S}'(\mathrm{x})$, *and*
- *for all* $\mathrm{X} \in \mathcal{D}(\mathcal{S}')$, $\mathcal{S}(\mathrm{X}) \succeq \mathcal{S}'(\mathrm{X})$.

Enrichment is a pre-order that defines a *subtyping* relation on semantic structures (i.e. $\mathcal{S}$ is a subtype of $\mathcal{S}'$ if and only if $\mathcal{S} \succeq \mathcal{S}'$).

Definition 2 (Functor Instantiation) *A semantic functor $\forall P.\mathcal{S} \to \mathcal{X}$ instantiates to a* functor instance *$\mathcal{S}' \to \mathcal{X}'$, written $\forall P.\mathcal{S} \to \mathcal{X} > \mathcal{S}' \to \mathcal{X}'$, if and only if $\varphi(\mathcal{S}) = \mathcal{S}'$ and $\varphi(\mathcal{X}) = \mathcal{X}'$, for some realisation φ with $\mathcal{D}(\varphi) = P$.*

Definition 3 (Signature Matching) *A semantic structure $\mathcal{S}$* matches *a signature $\Lambda P.\mathcal{S}'$ if and only if $\mathcal{S} \succeq \varphi(\mathcal{S}')$ for some realisation φ with $\mathcal{D}(\varphi) = P$.*

The static semantics of Mini-SML is defined by the denotation judgements in Fig. 5 that relate type phrases to their denotations, and the classification judgements in Fig. 6 that relate term phrases to their semantic types. A complete presentation and detailed explanation of these rules may be found in [18, 16, 17].

$$\boxed{\mathcal{C} \vdash \mathrm{u} \rhd u} \quad \frac{\mathrm{t} \in \mathcal{D}(\mathcal{C})}{\mathcal{C} \vdash \mathrm{t} \rhd \mathcal{C}(\mathrm{t})} \quad \frac{\mathcal{C} \vdash \mathrm{u} \rhd u \quad \mathcal{C} \vdash \mathrm{u}' \rhd u'}{\mathcal{C} \vdash \mathrm{u} \to \mathrm{u}' \rhd u \to u'} \quad \frac{}{\mathcal{C} \vdash \textbf{int} \rhd \mathrm{int}} \quad \frac{\mathcal{C} \vdash \mathrm{sp} : \mathcal{S} \quad \mathrm{t} \in \mathcal{D}(\mathcal{S})}{\mathcal{C} \vdash \mathrm{sp.t} \rhd \mathcal{S}(\mathrm{t})}$$

$$\boxed{\mathcal{C} \vdash \mathrm{B} \rhd \mathcal{L}} \quad \frac{\mathcal{C} \vdash \mathrm{u} \rhd u \quad \mathcal{C}, \mathrm{t} \rhd u \vdash \mathrm{B} \rhd \mathit{\Lambda} P.\mathcal{S} \quad \mathrm{t} \notin \mathcal{D}(\mathcal{S}) \quad P \cap \mathcal{V}(u) = \emptyset}{\mathcal{C} \vdash (\textbf{type}\ \mathrm{t} = \mathrm{u}; \mathrm{B}) \rhd \mathit{\Lambda} P.(\mathrm{t} \rhd u, \mathcal{S})}$$

$$\frac{\alpha \notin \mathcal{V}(\mathcal{C}) \quad \mathcal{C}, \mathrm{t} \rhd \alpha \vdash \mathrm{B} \rhd \mathit{\Lambda} P.\mathcal{S} \quad \mathrm{t} \notin \mathcal{D}(\mathcal{S}) \quad \alpha \notin P}{\mathcal{C} \vdash (\textbf{type}\ \mathrm{t}; \mathrm{B}) \rhd \mathit{\Lambda}\{\alpha\} \cup P.(\mathrm{t} \rhd \alpha, \mathcal{S})}$$

$$\frac{\mathcal{C} \vdash \mathrm{u} \rhd u \quad \mathcal{C}, \mathrm{x} : u \vdash \mathrm{B} \rhd \mathit{\Lambda} P.\mathcal{S} \quad \mathrm{x} \notin \mathcal{D}(\mathcal{S}) \quad P \cap \mathcal{V}(u) = \emptyset}{\mathcal{C} \vdash (\textbf{val}\ \mathrm{x} : \mathrm{u}; \mathrm{B}) \rhd \mathit{\Lambda} P.(\mathrm{x} : u, \mathcal{S})}$$

$$\frac{\mathcal{C} \vdash \mathrm{S} \rhd \mathit{\Lambda} P.\mathcal{S} \ \ P \cap \mathcal{V}(\mathcal{C}) = \emptyset \quad \mathcal{C}, \mathrm{X} : \mathcal{S} \vdash \mathrm{B} \rhd \mathit{\Lambda} Q.\mathcal{S}' \quad \mathrm{X} \notin \mathcal{D}(\mathcal{S}') \ \ Q \cap (P \cup \mathcal{V}(\mathcal{S})) = \emptyset}{\mathcal{C} \vdash (\textbf{structure}\ \mathrm{X} : \mathrm{S}; \mathrm{B}) \rhd \mathit{\Lambda} P \cup Q.(\mathrm{X} : \mathcal{S}, \mathcal{S}')}$$

$$\frac{}{\mathcal{C} \vdash \epsilon_{\mathrm{B}} \rhd \mathit{\Lambda}\emptyset.\epsilon_{\mathcal{S}}}$$

$$\boxed{\mathcal{C} \vdash \mathrm{S} \rhd \mathcal{L}} \quad \frac{\mathcal{C} \vdash \mathrm{B} \rhd \mathcal{L}}{\mathcal{C} \vdash \textbf{sig}\ \mathrm{B}\ \textbf{end} \rhd \mathcal{L}} \quad \frac{\mathrm{T} \in \mathcal{D}(\mathcal{C})}{\mathcal{C} \vdash \mathrm{T} \rhd \mathcal{C}(\mathrm{T})}$$

Fig. 5. Denotation Judgements

$$\boxed{\mathcal{C} \vdash \mathrm{e} : u} \quad \frac{\mathrm{x} \in \mathcal{D}(\mathcal{C})}{\mathcal{C} \vdash \mathrm{x} : \mathcal{C}(\mathrm{x})} \quad \frac{\mathcal{C} \vdash \mathrm{u} \rhd u \quad \mathcal{C}, \mathrm{x} : u \vdash \mathrm{e} : u'}{\mathcal{C} \vdash \lambda \mathrm{x} : \mathrm{u.e} : u \to u'} \quad \ldots \quad \frac{\mathcal{C} \vdash \mathrm{sp} : \mathcal{S} \quad \mathrm{x} \in \mathcal{D}(\mathcal{S})}{\mathcal{C} \vdash \mathrm{sp.x} : \mathcal{S}(\mathrm{x})}$$

$$\boxed{\mathcal{C} \vdash \mathrm{sp} : \mathcal{S}} \quad \frac{\mathrm{X} \in \mathcal{D}(\mathcal{C})}{\mathcal{C} \vdash \mathrm{X} : \mathcal{C}(\mathrm{X})} \quad \frac{\mathcal{C} \vdash \mathrm{sp} : \mathcal{S} \quad \mathrm{X} \in \mathcal{D}(\mathcal{S})}{\mathcal{C} \vdash \mathrm{sp.X} : \mathcal{S}(\mathrm{X})}$$

$$\boxed{\mathcal{C} \vdash \mathrm{b} : \mathcal{X}} \quad \frac{\mathcal{C} \vdash \mathrm{u} \rhd u \quad \mathcal{C}, \mathrm{t} \rhd u \vdash \mathrm{b} : \exists P.\mathcal{S} \quad P \cap \mathcal{V}(u) = \emptyset}{\mathcal{C} \vdash (\textbf{type}\ \mathrm{t} = \mathrm{u}; \mathrm{b}) : \exists P.(\mathrm{t} \rhd u, \mathcal{S})}$$

$$\frac{\mathcal{C} \vdash \mathrm{e} : u \quad \mathcal{C}, \mathrm{x} : u \vdash \mathrm{b} : \exists P.\mathcal{S} \quad P \cap \mathcal{V}(u) = \emptyset}{\mathcal{C} \vdash (\textbf{val}\ \mathrm{x} = \mathrm{e}; \mathrm{b}) : \exists P.(\mathrm{x} : u, \mathcal{S})}$$

$$\frac{\mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S} \quad P \cap \mathcal{V}(\mathcal{C}) = \emptyset \quad \mathcal{C}, \mathrm{X} : \mathcal{S} \vdash \mathrm{b} : \exists Q.\mathcal{S}' \quad Q \cap (P \cup \mathcal{V}(\mathcal{S})) = \emptyset}{\mathcal{C} \vdash (\textbf{structure}\ \mathrm{X}\ =\ \mathrm{s;b}) : \exists P \cup Q.(\mathrm{X} : \mathcal{S}, \mathcal{S}')}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash \mathrm{S} \rhd \mathit{\Lambda} P.\mathcal{S} \qquad P \cap \mathcal{V}(\mathcal{C}) = \emptyset \qquad \mathcal{C}, \mathrm{X} : \mathcal{S} \vdash \mathrm{s} : \mathcal{X} \\ \mathcal{C}, \mathrm{F} : \forall P.\mathcal{S} \to \mathcal{X} \vdash \mathrm{b} : \mathcal{X}'\end{array}}{\mathcal{C} \vdash (\textbf{functor}\ \mathrm{F}\ (\mathrm{X} : \mathrm{S})\ =\ \mathrm{s}; \mathrm{b}) : \mathcal{X}'}$$

$$\frac{\mathcal{C} \vdash \mathrm{S} \rhd \mathcal{L} \quad \mathcal{C}, \mathrm{T} \rhd \mathcal{L} \vdash \mathrm{b} : \mathcal{X}}{\mathcal{C} \vdash (\textbf{signature}\ \mathrm{T}\ =\ \mathrm{S};\ \mathrm{b}) : \mathcal{X}} \qquad \frac{}{\mathcal{C} \vdash \epsilon_{\mathrm{b}} : \exists\emptyset.\epsilon_{\mathcal{S}}}$$

$$\boxed{\mathcal{C} \vdash \mathrm{s} : \mathcal{X}} \quad \frac{\mathcal{C} \vdash \mathrm{sp} : \mathcal{S}}{\mathcal{C} \vdash \mathrm{sp} : \exists\emptyset.\mathcal{S}} \quad \frac{\mathcal{C} \vdash \mathrm{b} : \mathcal{X}}{\mathcal{C} \vdash \textbf{struct}\ \mathrm{b}\ \textbf{end} : \mathcal{X}}$$

$$\frac{\mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S} \quad P \cap \mathcal{V}(\mathcal{C}(\mathrm{F})) = \emptyset \quad \mathcal{C}(\mathrm{F}) > \mathcal{S}' \to \exists Q.\mathcal{S}'' \quad \mathcal{S} \succeq \mathcal{S}' \quad Q \cap P = \emptyset}{\mathcal{C} \vdash \mathrm{F}(\mathrm{s}) : \exists P \cup Q.\mathcal{S}''}$$

$$\frac{\mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S} \quad \mathcal{C} \vdash \mathrm{S} \rhd \mathit{\Lambda} Q.\mathcal{S}' \quad P \cap \mathcal{V}(\mathit{\Lambda} Q.\mathcal{S}') = \emptyset \quad \mathcal{S} \succeq \varphi(\mathcal{S}') \quad \mathcal{D}(\varphi) = Q}{\mathcal{C} \vdash (\mathrm{s} :> \mathrm{S}) : \exists Q.\mathcal{S}'}$$

Fig. 6. Classification Judgements (some rules for Core expressions omitted)

5 Package Types

The motivation for introducing first-class structures is to extend the range of computations on structures. One way to do this is to extend structure expressions, and thus computation at the Modules level, with the general-purpose computational constructs usually associated with the Core. Instead of complicating the Modules language in this way, we propose to maintain the distinction between Core and Modules, but relax the stratification. Our proposal is to extend the Core language with a family of Core types, called *package types*, corresponding to first-class structures. A package type is introduced by encapsulating, or *packing*, a structure as a Core value. A package type is eliminated by breaking an encapsulation, *opening* a Core value as a structure in the scope of another Core expression. Because package types are ordinary Core types, packages are first-class citizens of the Core. The introduction and elimination phrases allow computation to alternate between computation at the level of Modules and computation at the level of the Core, without having to identify the notions of computation.

Our extension requires just three new syntactic constructs, all of which are additions to the Core language:

Core Types	u ::=. . .	| <S>	package type
Core Expressions	e ::=. . .	| **pack** s **as** S	package introduction
		| **open** e **as** X : S **in** e′	package elimination

The syntactic Core type <S>, which we call a *package type*, denotes the type of a Core expression that evaluates to an encapsulated structure value. The actual type of this structure value must match the signature S: i.e. if S denotes $\Lambda P.\mathcal{S}$, then the type of the encapsulated structure must be a subtype of $\varphi(\mathcal{S})$, for φ a realisation with $\mathcal{D}(\varphi) = P$. Two package types <S> and <S′> will be equivalent if their denotations (not just their syntactic forms) are equivalent.

The Core expression **pack** s **as** S introduces a value of package type <S>. Assuming a call-by-value dynamic semantics, the phrase is evaluated by evaluating the structure expression s and encapsulating the resulting structure value as a Core value. The static semantics needs to ensure that the type of the structure expression matches the signature S. Note that two expressions **pack** s **as** S and **pack** s′ **as** S may have the same package type <S> even when the actual types of s and s′ differ (i.e. the types both match the signature, but in different ways).

The Core expression **open** e **as** X : S **in** e′ eliminates a value of package type <S>. Assuming a call-by-value dynamic semantics, the expression e is evaluated to an encapsulated structure value, this value is bound to the structure identifier X, and the value of the entire phrase is obtained by evaluating the client expression e′ in the extended environment. The static semantics needs to ensure that e has the package type <S> and that the type of e′ does not vary with the actual type of the encapsulated structure X.

The semantic Core types of Mini-SML must be extended with the semantic counterpart of syntactic package types. In Mini-SML, the type of a structure expression is an existential structure $\mathcal{X}$ determined by the judgement form $\mathcal{C} \vdash$

s : $\mathcal{X}$. Similarly, the denotation of a package type, which describes the type of an encapsulated structure value, is just an encapsulated existential structure:

$$u \in \mathit{Type} ::= \ldots \mid \,<\!\mathcal{X}\!> \qquad \text{semantic package type}$$

We identify semantic package types that are equivalent up to matching:

Definition 4 (Equivalence of Semantic Package Types) *Two semantic package types* $<\!\exists P.\mathcal{S}\!>$ *and* $<\!\exists P'.\mathcal{S}'\!>$ *are* equivalent *if, and only if,*

- $P' \cap \mathcal{V}(\exists P.\mathcal{S}) = \emptyset$ *and* $\mathcal{S}' \succeq \varphi(\mathcal{S})$ *for some realisation* φ *with* $\mathcal{D}(\varphi) = P$*;*
- $P \cap \mathcal{V}(\exists P'.\mathcal{S}') = \emptyset$ *and* $\mathcal{S} \succeq \varphi'(\mathcal{S}')$ *for some realisation* φ' *with* $\mathcal{D}(\varphi') = P'$*.*

The following rules extend the Core judgements $\mathcal{C} \vdash \mathrm{u} \rhd u$ and $\mathcal{C} \vdash \mathrm{e} : u$:

$$\frac{\mathcal{C} \vdash \mathrm{S} \rhd \Lambda P.\mathcal{S}}{\mathcal{C} \vdash \,<\!\mathrm{S}\!>\, \rhd \,<\!\exists P.\mathcal{S}\!>} \tag{1}$$

Rule 1 relates a syntactic package type to its denotation as a semantic package type. The parameters of the semantic signature $\Lambda P.\mathcal{S}$ stem from opaque type specifications in S and determine the quantifier of the package type $<\!\exists P.\mathcal{S}\!>$.

$$\frac{\mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S} \quad \mathcal{C} \vdash \mathrm{S} \rhd \Lambda Q.\mathcal{S}' \quad P \cap \mathcal{V}(\Lambda Q.\mathcal{S}') = \emptyset \quad \mathcal{S} \succeq \varphi(\mathcal{S}') \quad \mathcal{D}(\varphi) = Q}{\mathcal{C} \vdash (\mathbf{pack}\ \mathrm{s}\ \mathbf{as}\ \mathrm{S}) : \,<\!\exists Q.\mathcal{S}'\!>} \tag{2}$$

Rule 2 is the introduction rule for package types. Provided s has existential type $\exists P.\mathcal{S}$ and S denotes the semantic signature $\Lambda Q.\mathcal{S}'$, the existential quantification over P is eliminated in order to verify that $\mathcal{S}$ matches the signature. The side condition $P \cap \mathcal{V}(\Lambda Q.\mathcal{S}') = \emptyset$ prevents the capture of free variables in the signature by the bound variables in P and ensures that these variables are treated as hypothetical types. The semantic signature $\Lambda Q.\mathcal{S}'$ describes a family of semantic structures and the requirement is that the type $\mathcal{S}$ of the structure expression enriches, i.e. is a subtype of, some member $\varphi(\mathcal{S}')$ of this family. In the resulting package type $<\!\exists Q.\mathcal{S}'\!>$, the existential quantification over Q hides the actual realisation, rendering type components specified opaquely in S abstract. Because the rule merely requires that $\mathcal{S}$ is a subtype of $\varphi(\mathcal{S}')$, the package **pack** s **as** S may have fewer components than the actual structure s.

$$\frac{\mathcal{C} \vdash \mathrm{e} : \,<\!\exists P.\mathcal{S}\!> \quad \mathcal{C} \vdash \mathrm{S} \rhd \Lambda P.\mathcal{S} \;\; P \cap \mathcal{V}(\mathcal{C}) = \emptyset \quad \mathcal{C}, \mathrm{X} : \mathcal{S} \vdash \mathrm{e}' : u \;\; P \cap \mathcal{V}(u) = \emptyset}{\mathcal{C} \vdash (\mathbf{open}\ \mathrm{e}\ \mathbf{as}\ \mathrm{X} : \mathrm{S}\ \mathbf{in}\ \mathrm{e}') : u} \tag{3}$$

Rule 3 is the elimination rule for package types. Provided e has package type $<\!\exists P.\mathcal{S}\!>$, where this type is determined by the denotation of the explicit syntactic signature S, the client e$'$ of the package is classified in the extended context $\mathcal{C}, \mathrm{X} : \mathcal{S}$. The side-condition $P \cap \mathcal{V}(\mathcal{C}) = \emptyset$ prevents the capture of free variables in $\mathcal{C}$ by the bound variables in P and ensures that these variables are treated as hypothetical types for the classification of e$'$. By requiring that e$'$ is

```
structure Sieve =
 struct type nat = TwoOnwards.nat; type state = <Stream>;
        val start = pack TwoOnwards as Stream;
        val next = λs:state.open s as S:Stream in pack Next(S) as Stream;
        val value = λs:state.open s as S:Stream in S.value S.start
 end;
val nthstate = fix λnthstate:int->Sieve.state.
                   λn:int.ifzero n then Sieve.start
                          else Sieve.next (nthstate (pred n));
val nthprime = λn:int.Sieve.value (nthstate n);
```

Fig. 7. The Sieve implemented using package types.

polymorphic in P, the actual realisation of these hypothetical types is allowed to vary with the value of e. Moreover, because $\mathcal{S}$ is a generic structure matching the signature S, the rule ensures that e′ does not access any components of X that are not specified in S: thus the existence of any unspecified components is allowed to vary with the actual value of e. Finally, the side condition $P \cap \mathcal{V}(u) = \emptyset$ prevents any variation in the actual realisation of P from affecting the type of the phrase.

Observe that the explicit signature S in the term **open** e **as** X : S **in** e′ uniquely determines the Core type of the expression e. This is significant for an implicitly typed language like Standard ML's Core: the explicit signature ensures that the type inference problem for that Core remains tractable and has principal solutions. Intuitively, the type inference algorithm [16] never has to *guess* the type of an expression that is used as a package. The explicit signature in the term **pack** s **as** S ensures that the package type of the expression corresponds to a well-formed signature (this may not be the case for the actual type of s): testing the equivalence of such well-formed package types (even modulo unification) can be performed by two appeals to a signature matching algorithm [16].

Rules 2 and 3 are closely related to the standard rules for second-order existential types in Type Theory [12]. The main difference, aside from manipulating n-ary, not just unary, quantifiers is that these rules also mediate between the universe of Module types and the universe of Core types. [16, 18] sketch proofs of type soundness for package types; [18] discusses implementation issues.

6 The Sieve Revisited

The addition of package types allows us to define the structure `Sieve` implementing the Sieve of Eratosthenes (Fig. 7). The Core type `Sieve.state` is the type of packaged streams `<Stream>`. The Core value `Sieve.start` is the packaged stream `TwoOnwards`. The Core function `Sieve.next` returns the next state of `Sieve` by opening the supplied state, sifting the encapsulated stream, and packaging the resulting stream as a Core value. The Core function `Sieve.value` returns the first value of its encapsulated stream argument.

It is easy to verify that `Sieve` has type:

$$\exists\emptyset.(\texttt{nat}\triangleright \text{int}, \texttt{state}\triangleright u, \texttt{start}\!: u, \texttt{next}\!: u \to u, \texttt{value}\!: u \to \text{int}),$$

where $u \equiv <\exists\{\alpha\}.(\texttt{nat}\triangleright \text{int}, \texttt{state}\triangleright \alpha, \texttt{start}\colon \alpha, \texttt{next}\colon \alpha \to \alpha, \texttt{value}\colon \alpha \to \text{int})>$ is the type of packed streams.

`Sieve` elegantly captures the impredicative description of the Sieve as a *stream* constructed from streams: its type also matches `Stream`, since

$$\begin{aligned}&(\texttt{nat}\triangleright \text{int}, \texttt{state}\triangleright u, \texttt{start}\colon u, \texttt{next}\colon u \to u, \texttt{value}\colon u \to \text{int}) \succeq \\ &\qquad \{\alpha \mapsto u\}\,(\texttt{nat}\triangleright \text{int}, \texttt{state}\triangleright \alpha, \texttt{start}\colon \alpha, \texttt{next}\colon \alpha \to \alpha, \texttt{value}\colon \alpha \to \text{int}).\end{aligned}$$

`Sieve` is a useful implementation because it allows us to define the functions `nthstate` and `nthprime` of Fig. 7. Since the states of `Sieve` are just ordinary Core values, which happen to have package types, the function `nthstate` n can use recursion on n to construct the n^{th} state of `Sieve`. In turn, this permits the function `nthprime` n to calculate the n^{th} prime, for an *arbitrary* n. Recall that, in the absence of package types, these functions could not be defined using the implementation of the Sieve we gave in Section 3.

7 Another Example: Dynamically-Sized Arrays

Package types permit the actual realisation of an abstract type to depend on the result of a Core computation. For this reason, package types strictly extend the class of abstract types that can be defined in vanilla Mini-SML.

A familiar example of such a type is the type of *dynamically* allocated arrays of size n, where n is a value that is computed at run-time. For simplicity, we implement *functional* arrays of size 2^n, for arbitrary $n \geq 0$ (Fig. 8).

The signature `Array` specifies structures implementing integer arrays with the following interpretation. For a fixed n, the type `array` represents arrays containing 2^n entries of type `entry` (equivalent to `int`). The function `init` e creates an array that has its entries initialised to the value of e. The function `sub` a i returns the value of the (i mod 2^n)-th entry of the array a. The function `update` a i e returns an array that is equivalent to the array a, except for the (i mod 2^n)-th entry that is updated with the value of e. Interpreting each index i modulo 2^n allows us to omit array bound checks.

The structure `ArrayZero` implements arrays of size $2^0 = 1$. An array is represented by its sole entry with trivial `init`, `sub` and `update` functions.

The functor `ArraySucc` maps a structure `A`, implementing arrays of size 2^n, to a structure implementing arrays of size 2^{n+1}. The functor represents an array of size 2^{n+1} as a pair of arrays of size 2^n. Entries with even (odd) indices are stored in the first (second) component of the pair. The function `init` e returns a pair of initialised arrays of size 2^n. The function `sub` a i (`update` a i e) uses the parity of i to determine which subarray to subscript (update).

The Core function `mkArray` n uses recursion on n to construct a package implementing arrays of size 2^n. Notice that the actual realisation of the abstract type `array` returned by `mkArray` n is a balanced, nested cross product of depth n: the shape of this type depends on the run-time value of n. Interestingly, this is an example of "data-structural bootstrapping" [14] yet does not use non-regular recursive types or polymorphic recursion: it does not even use recursive types!

```
signature Array = sig type entry = int; type array; (*array is opaque*)
                      val init: entry → array;
                      val sub: array → int → entry;
                      val update: array → int → entry → array
                  end;
structure ArrayZero = struct type entry = int; type array = entry;
                             val init = λe:entry.e;
                             val sub = λa:array.λi:int.a;
                             val update = λa:array.λi:int.λe:entry.e
                      end;
functor ArraySucc (A:Array) =
   struct type entry = A.entry; type array = A.array * A.array;
          val init = λe:entry. (A.init e, A.init e)
          val sub = λa:array.λi:int.
               ifzero mod i 2 then A.sub (fst a) (div i 2)
                              else A.sub (snd a) (div i 2);
          val update = λa:array.λi:int.λe:entry.
               ifzero mod i 2 then (A.update (fst a) (div i 2) e, snd a)
                              else (fst a, A.update (snd a) (div i 2) e)
   end;
val mkArray = fix λmkArray:int→<Array>.
               λn:int. ifzero n then pack ArrayZero as Array
                       else open mkArray (pred n) as A:Array in
                                 pack ArraySucc(A) as Array;
```

Fig. 8. `mkArray` n returns an abstract implementation of arrays of size 2^n.

8 Contribution

For presentation purposes, we restricted our attention to an explicitly typed, monomorphic Core language and a first-order Modules language. In [16], we demonstrate that the extension with package types may also be applied to a Standard ML-like Core language that supports the definition of type constructors and implicitly typed, polymorphic values. For instance, Section 7.3 of [16] generalises the example of Section 7 to an implementation of dynamically sized *polymorphic* arrays where `array` is a unary type constructor taking the type of entries as an argument and the array operations are suitably polymorphic. Moreover, this extension is formulated with respect to a higher-order Modules calculus that allows functors, not just structures, to be treated as first class citizens of the Modules language and, via package types, the Core language too. This proposal is practical: we present a well-behaved algorithm that integrates type inference for the extended Core with type checking for higher-order Modules. First-class and higher-order modules are available in Moscow ML V2.00[15].

Our approach to obtaining first-class structures is novel because it leaves the Modules language unchanged, relies on a simple extension of the Core language only and avoids introducing subtyping in the Core type system, which would otherwise pose severe difficulties for Core-ML type inference. (Although Mitchell *et al.* [11, 5] first suggested the idea of coercing a structure to a first-class existential type they did not require explicit introduction and elimination terms: our insistence on these terms enables Core-ML type inference.) Our work refutes Harper and Mitchell's claim [2] that the existing type structure of Standard ML cannot accommodate first-class structures without sacrificing the compile-time/run-time phase distinction and decidable type checking. This is a limitation of their proposed model, which is based on first-order dependent types, but does not transfer to the simpler, second-order type theory [17] of Standard ML.

Our motivation for introducing first-class structures was to extend the range of computations on structures. One way to achieve this is to extend structure expressions directly with computational constructs usually associated with the Core. Taken to the extreme, this approach relaxes the stratification between Modules and the Core by removing the distinction between them, amalgamating both in a single language. This is the route taken by Harper and Lillibridge [1, 8]. Unfortunately, the identification of Core and Modules types renders subtyping, and thus type checking, undecidable. Leroy [6] briefly considers this approach without formalising it but observes that the resulting interaction between Core and Modules computation violates the type soundness of *applicative* functors [7]. Odersky and Läufer [13] and Jones [4] adopt a different tack and extend implicitly typed Core-ML with impredicative type quantification and higher-order type constructors that can model some, but not all, of the features of Standard ML Modules while providing first-class and higher-order modules.

Our approach is different. We maintain the distinction between Core and Modules, but relax the stratification by extending the Core language with package types. The introduction and elimination phrases for package types allow computation to alternate between computation at the level of Modules and computation at the level of the Core, without having to identify the notions of computation. This is reflected in the type system in which the Modules and Core typing relations are distinct. This is a significant advantage for implicitly typed Core languages like Core-ML. At the Modules level, the explicitly typed nature of Modules makes it possible to accommodate subtyping, functors with polymorphic arguments and true type constructors in the type checker for the typing relation. At the Core-ML level, the absence of subtyping, the restriction that ML functions may only take monomorphic arguments and that ML type variables range over types (but not type constructors) permits the use of Hindley-Milner [3, 9] type inference. In comparison, the amalgamated languages of [1, 8] support type constructors and subtyping, but at the cost of an explicitly typed Core fragment; [13, 4] support partial type inference, but do not provide subtyping on structures, type components in structures or a full treatment of type constructors, whose expansion and contraction must be mediated by explicit Core terms instead of implicit β-conversion.

Although not illustrated here, the advantage of distinguishing between Modules computation and Core computation is that they can be designed to satisfy different invariants [16]. For instance, the invariant needed to support applicative functors [7, 16], namely that the abstract types returned by a functor depend only on its type arguments and not the value of its term argument, is violated if we extend Modules computation directly with general-purpose computational constructs. Applicative functors provide better support for programming with higher-order Modules; general-purpose constructs are vital for a useful Core. In [16], we show that maintaining the separation between Modules and Core computation accommodates both applicative functors and a general-purpose Core, without violating type soundness. Type soundness is preserved by the addition of package types, because these merely extend the computational power of the Core, not Modules (package elimination is weaker than including Core expressions in Module expressions). The languages of [1, 8] have higher-order functors, but their single notion of computation implies a trade-off between supporting either *applicative* functors or general-purpose computation. Since ruling out the latter is too restrictive, the functors of these calculi are not applicative.

References

[1] R. Harper, M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *21st ACM Symp. Principles of Prog. Lang.*, 1994.

[2] R. Harper, J. C. Mitchell. On the type structure of Standard ML. In *ACM Trans. Prog. Lang. Syst.*, volume 15(2), pages 211–252, 1993.

[3] J. R. Hindley. The principal type scheme of an object in combinatory logic. *Trans. of the American Mathematical Society*, 146:29-40, 1969.

[4] M. Jones. Using Parameterized Signatures to Express Modular Structure. In *23rd ACM Symp. Principles of Prog. Lang.*, 1996.

[5] D. Katiyar, D. Luckham, J. Mitchell. A type system for prototyping languages. In *24th ACM Symp. Principles of Prog. Lang.*, 1994.

[6] X. Leroy. Manifest types, modules, and separate compilation. In *Proc. 21st Symp. Principles of Prog. Lang.*, pages 109–122. ACM Press, 1994.

[7] X. Leroy. Applicative functors and fully transparent higher-order modules. In *Proc. 22nd Symp. Principles of Prog. Lang.*, pages 142–153. ACM Press, 1995.

[8] M. Lillibridge. Translucent Sums: A Foundation for Higher-Order Module Systems. PhD thesis, School of Computer Science, Carnegie Mellon University, 1997.

[9] R. Milner. A theory of type polymorphism in programming languages. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[10] R. Milner, M. Tofte, R. Harper, D. MacQueen. The Definition of Standard ML (Revised). MIT Press, 1997.

[11] J. C. Mitchell, S. Meldal, N. Madhav. An extension of Standard ML modules with subtyping and inheritance. In *18th ACM Symp. Principles of Prog. Lang.*, 1991.

[12] J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

[13] M. Odersky, K. Läufer. Putting Type Annotations To Work In *23rd ACM Symp. Principles of Prog. Lang.*, 1996.

[14] C. Okasaki. Purely Functional Data Structures. Cambridge University Press, 1998.

[15] S. Romanenko, P. Sestoft. Moscow ML. (`www.dina.kvl.dk/~sestoft/mosml`).

[16] C. V. Russo. Types For Modules. PhD Thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 1998.

[17] C. V. Russo. Non-Dependent Types For Standard ML Modules. In *1999 Int'l Conf. on Principles and Practice of Declarative Programming.*

[18] C. V. Russo. First-Class Structures for Standard ML (long version). Forthcoming Technical Report, LFCS, Division of Informatics, University of Edinburgh, 2000.

Constraint-Based Inter-Procedural Analysis of Parallel Programs

Helmut Seidl[1] and Bernhard Steffen[2]

[1] FB IV – Informatik, Universität Trier,
D-54286 Trier, Germany,
seidl@uni-trier.de

[2] Lehrstuhl für Programmiersysteme, Universität Dortmund,
Baroper Straße 301, D-44221 Dortmund, Germany,
Bernhard.Steffen@cs.uni-dortmund.de

Abstract. We provide a uniform framework for the analysis of programs with procedures and explicit, unbounded, fork/join parallelism covering not only bitvector problems like reaching definitions or live variables but also non-bitvector problems like simple constant propagation. Due to their structural similarity to the sequential case, the resulting algorithms are as efficient as their widely accepted sequential counterparts, and they can easily be integrated in existing program analysis environments like e.g. MetaFrame or PAG. We are therefore convinced that our method will soon find its way into industrial-scale computer systems.

Keywords: Inter-procedural program analysis, explicit parallelism, bitvector problems, simple constant propagation, coincidence theorems.

1 Introduction

The analysis of parallel programs is known as a notoriously hard problem. Even without procedures and with only bounded parallelism the analysis typically suffers from the so-called *state explosion problem*: in general, already the required control structures grow exponentially with the number of parallel components. Bitvector analyses, dominant in most practical compilers, escape this problem in the context of fork/join-parallelism [11, 9]: a simple pre-process is sufficient to adapt sequential intra-procedural bitvector analyses to directly work on *parallel flow graphs* which concisely and explicitly represent the program's parallelism. Key for this adaptation was to change from a *property* analysis (directly associating program points with properties) to an *effect* analysis[1] associating program points with a *property transformer* resembling the effect of the 'preceding' program fragment. The simplicity of the adaption results from the fact that bitvector analyses can conceptually be "sliced" into separate analyses for each individual bit-component each of which only requires the consideration of a three-point transformer domain.

In order to handle also procedures and unbounded parallelism, Esparza and Knoop observed that the described problem profile also admits an automata

[1] Second-order analysis in the terminology of [11].

G. Smolka (Ed.): ESOP/ETAPS 2000, LNCS 1782, pp. 351–365, 2000.

theoretic treatment [5]. This observation has been carefully developed by Esparza and Podelski in [6]. The resulting algorithm requires involved concepts, like e.g. tree automata, and, from a program analyzer's perspective, the results are rather indirect: the reachability analysis computes regular sets characterizing the set of states satisfying a particular property. More precisely, the algorithm treats each bit-component of the analysis separately. For each such component an automata construction is required which is linear in the *product* of the size of the program and the size of an automaton describing reachable configurations. The latter automaton can grow linearly in the size of the program as well – implying that the analysis of each component is at least *quadratic* in the program size.

In this paper we present a much more direct framework for the inter-procedural analysis of fork/join parallel programs. We propose a constraint-based approach which naturally arises from an algebraic reformulation of the intra-procedural method presented in [11, 9]. Our approach closely resembles the classical understanding of bitvector analysis, has a complexity which is *linear* in the program size and admits elegant, algebraic proofs. Summarizing, we contribute to the state of the art by

1. Providing a uniform characterization of the captured analysis profile which *simultaneously* addresses all involved program entities, e.g., all program variables at once for live variable analysis or all program expressions at once for availability of expressions. Moreover, this profile goes beyond pure bitvector analyses as it e.g. also captures simple constant propagation [9].
2. Basing our development on a constraint characterization of valid parallel execution paths: the constraint system for the actual analyses simply results from an abstract interpretation [3, 4, 2] of this characterization.
3. Presenting a framework which supports algebraic reasoning. E.g., the proof for proposition 2(3) – resembling the central Main Lemma of [11] – straightforwardly evolves from our profile characterization.
4. Guaranteeing essentially the same performance as for purely inter-procedural bitvector analyses by exploiting the results of a generalized *possible interference* analysis [11].

As a consequence, the presented framework is tightly tailored for the intended application area. It directly associates the program points with the required information based on classical constraint solving through (e.g., worklist based) fixpoint iteration. This can be exploited to obtain simple implementations in current program analysis generators like DFA& OPT MetaFrame [10] or PAG [1], which provide all the required fixpoint iteration machinery.

The paper is organized as follows. After formally introducing explicitly parallel programs with procedures in section 2, we define the notion of parallel execution paths in section 3, and specify our analysis problem in section 4. Section 5 then presents a precise effect analysis for procedures, which is the basis for the precise inter-procedural reachability analysis given in section 6. Finally, section 7 discusses possible extensions of our formal development, while section 8 gives our conclusions and perspectives.

2 Programs as Control-Flow Graphs

We assume that programs are given as (annotated) control-flow graphs (cfg's for short). An edge in the cfg either is a call of a single procedure, a parallel call to two procedures, or a basic computation step. An example of such a cfg is given in figure 1. There, we only visualized the annotation of call and parallel call edges. Observe that this cfg indeed introduces an unbounded number of instances of procedure q running in parallel.

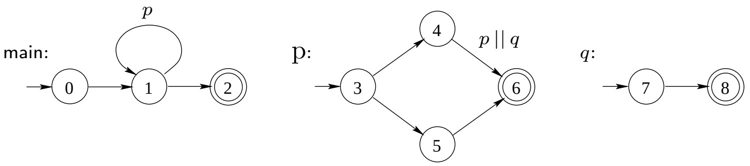


Fig. 1. An Example Control-flow Graph.

Formally, a control-flow graph $\mathcal{G}$ for a program with procedures and explicit parallelism consists of a finite set Proc of procedures together with a collection $\mathcal{G}_p, p \in \mathsf{Proc}$, of disjoint *intra-procedural* control-flow graphs. We assume that there is one special procedure main with which program execution starts. The intra-procedural control-flow graph $\mathcal{G}_p$ of a procedure p consists of:

- A set N_p of *program points*;
- A special *entry point* $s \in N_p$ as well as a special *return point* $r \in N_p$;
- A set of edges $E_p \subseteq N_p \times N_p$;
- A subset $C_p \subseteq E_p$ of *call edges* where for $e \in C_p$, $\mathsf{call}\, e = p$ denotes that edge e calls the procedure p; and finally,
- A subset $P_p \subseteq E_p$ of *parallel call edges* where for $e \in P_p$, $\mathsf{call}\, e = p_1 \,||\, p_2$ denotes that edge e calls the procedures p_1 and p_2 in parallel.

Edges which are not contained in C_p or P_p are also called *basic edges.*

Practical Remark: It is just for convenience that we allow only binary parallelism in our programs. Our methods can be easily adapted to work also for more procedures being called in parallel or even parallel do-loops.
Also note that we do not consider synchronization between parallel threads by barriers or semaphores. Such constructs *limit* the amount of possible execution paths. By ignoring these, we may get *more* possible execution paths and thus (perhaps less precise but) still *safe* analysis results.

3 Parallel Execution Paths

The semantics of a parallel program is determined w.r.t. the set of parallel execution paths. What we are now going to formalize is an *interleaving semantics*

for parallely executable threads. We need the following auxiliary definitions. Let E denote a finite set of edges. Let $w = e_1 \ldots e_n$ be a word from E^* and $I = \{i_1 < \ldots < i_k\} \subseteq \{1, \ldots, n\}$ be a subset of positions in w. Then the *restriction* of w to I is given by $w|_I = e_{i_1} \ldots e_{i_k}$.
The *interleaving* of subsets $M_1, M_2 \subseteq E^*$ is defined by

$$M_1 \otimes M_2 = \{w \in E^* \mid \exists I_1 + I_2 = \{1, \ldots, |w|\} : w|_{I_1} \in M_1 \text{ and } w|_{I_2} \in M_2\}$$

Here, "+" denotes disjoint union of sets. Thus, $M_1 \otimes M_2$ consists of all possible interleavings of sequences from M_1 and M_2. Furthermore for $M \subseteq E^*$, let $\mathrm{pre}(M)$ denote the set of all prefixes of words in M, i.e.,

$$\mathrm{pre}(M) = \{u \in E^* \mid \exists v \in E^* : uv \in M\}$$

We consider the following sets of possible execution paths:

- For $p \in \mathsf{Proc}$, the set $\Pi(p)$ of all execution paths for p;
- For program point v of procedure p, the set $\Pi(v)$ of all paths starting at the entry point of p and reaching v on the *same level* (see below);
- For every procedure p, the set $\Pi_r(p)$ of all paths starting at from a call of main and *reaching* some call of p;
- For every program point v, the set $\Pi_r(v)$ of all paths starting at from a call of main and *reaching* program point v.

These sets are given through the least solutions of the following constraint systems (whose variables for simplicity are denoted by $\Pi(p), \Pi(v), \Pi_r(p), \Pi_r(v)$ as well). Let us start with the defining constraint system for the sets of same-level execution paths.

$$\begin{array}{lll}
\Pi(p) \supseteq \Pi(r) & r \text{ return point of } p & (1)\\
\Pi(s) \supseteq \{\epsilon\} & s \text{ entry point of a procedure} & (2)\\
\Pi(v) \supseteq \Pi(u) \cdot \{e\} & e = (u, v) \text{ basic edge} & (3)\\
\Pi(v) \supseteq \Pi(u) \cdot \Pi(p) & e = (u, v) \text{ calls } p & (4)\\
\Pi(v) \supseteq \Pi(u) \cdot (\Pi(p_1) \otimes \Pi(p_2)) & e = (u, v) \text{ calls } p_1 \,||\, p_2 & (5)
\end{array}$$

Lines (1) through (4) are the standard lines to determine the sets of all same-level execution paths as known from inter-procedural analysis of sequential programs. Line (1) says that the set of execution paths of procedure p is the set of same-level paths reaching the return point of p. Line (2) says that at least ϵ is a same-level execution path that reaches the entry point of a procedure. Line (3) says that for every basic edge $e = (u, v)$, the set of same-level execution paths reaching the program point v subsumes all same-level execution paths to u extended by e. Line (4) says that for every edge $e = (u, v)$ calling a procedure p, the set of same-level execution paths reaching the program point v subsumes all same-level execution paths reaching u extended by any execution path through the procedure p. Line (5) for a parallel call of $p_1 \,||\, p_2$ has the same form as line (4). But now the same-level execution paths to the program point before the call are extended by all interleavings of execution paths for p_1 and p_2.

In order to specify the sets $\Pi_r(p), \Pi_r(v)$, let us introduce the auxiliary sets $\Pi(v,p)$, v a program point, p a procedure, which give the sets of execution paths reaching v from a call of p. These auxiliary sets are defined as the least solution of the following system of constraints:

$$\begin{array}{lll} \Pi(v,q) \supseteq \Pi(v) & v \text{ program point of procedure } q & (1) \\ \Pi(v,q) \supseteq \Pi(u) \cdot \Pi(v,p) & e = (u,\_) \text{ calls } p \text{ in } q & (2) \\ \Pi(v,q) \supseteq \Pi(u) \cdot (\Pi(v,p_i) \otimes M) & e = (u,\_) \text{ calls } p_1 \,||\, p_2 \text{ in } q & (3) \end{array}$$

where M in line (3) is given by $M = \text{pre}(\Pi(p_{3-i}))$. The intuition behind this definition is as follows. Line (1) says that whenever v is a program point of procedure q, then the set of execution paths from q to v subsumes all same-level execution paths from q to v. Line (2) says that whenever at some edge $e = (u,\_)$ in the body of procedure q, some procedure p is called, then the set of execution paths from q to v subsumes all computation paths consisting of a same-level execution path from q to the program point u followed by an execution path from p to v. Finally, line (3) considers an edge $e = (u,\_)$ in the body of q which is a parallel call of p_1 and p_2. Then we have to append to the same-level execution paths to u all interleavings of execution paths from p_i to v with prefixes of same-level execution paths for the parallel procedure.
Given the $\Pi(v,q)$, we define the values $\Pi_r(v), \Pi_r(p)$ as the least solution of:

$$\begin{array}{ll} \Pi_r(v) \supseteq \Pi(v, \mathsf{main}) & v \text{ a program point} \\ \Pi_r(p) \supseteq \Pi_r(u) & \text{edge } (u,\_) \text{ calls } p,\ p \,||\, \_ \text{ or } \_ \,||\, p \end{array}$$

For now, let us assume that all the sets of execution paths $\Pi(v), \Pi_r(v), \Pi(p), \Pi_r(p)$ are non-empty. In section 7 we will explain how this assumption can be removed.

4 Semantics

Let $\mathbb{D}$ denote a complete lattice and $\mathbb{F} \subseteq \mathbb{D} \to \mathbb{D}$ a subset of monotonic functions from $\mathbb{D}$ to $\mathbb{D}$ which contains $\lambda x.\bot$ (the constant $\bot$-function) and $I = \lambda x.x$ (the identity) and is closed under composition "$\circ$" and least upper bounds. While $\mathbb{D}$ is meant to specify the set of abstract properties, $\mathbb{F}$ describes all possible ways how properties may be transformed when passing from one program point to the other. In this paper we make the following additional assumption:

- $\mathbb{D}$ is *distributive*, i.e., $a \sqcup (b \sqcap c) = (a \sqcap b) \sqcup (a \sqcap c)$ holds for all $a, b, c \in \mathbb{D}$;
- $\mathbb{D}$ has height $h < \infty$, i.e., every ascending chain of elements in $\mathbb{D}$ has length at most $h + 1$;
- set $\mathbb{F}$ consists of all functions of the form $f\,x = (a \sqcap x) \sqcup b$ with $a, b \in \mathbb{D}$.

Since $\mathbb{D}$ is distributive, all functions f in $\mathbb{F}$ are distributive as well, i.e., $f\,(a \sqcup b) = (f\,a) \sqcup (f\,b)$ for all $a, b \in \mathbb{D}$. Let us also mention that neither $\mathbb{D}$ nor $\mathbb{F}$ is demanded to be finite. However, since $\mathbb{D}$ has height h, the lattice $\mathbb{F}$ has height at most $2h$. The most prominent class of problems that satisfy our restrictions are *bitvector problems* like available expressions, reaching definitions, life variables or very busy expressions [7]. In these cases, we may choose $\mathbb{D} = \mathbb{B}^h$ where $\mathbb{B} = \{0 \sqsubset 1\}$.

There are, however, further analysis problems which meet our assumptions without being bitvector problems. This is the case, e.g., for simple constant propagation. Simple constant propagation tries to determine whether or not some constant has been assigned to a variable which later-on remains unchanged. For this application, we may choose $\mathbb{D} = V \to \mathbb{B}$ where V is the set of program variables and $\mathbb{B}$ is the flat lattice of possible values for program variables. Thus, an abstract value $d \in \mathbb{D}$ represents an assignment of variables to values. In particular, $\mathbb{D}$ has height $h = 2 \cdot \#V$. Note furthermore that for simple constant propagation, all functions $f \in \mathbb{F}$ are of the special form $f = \lambda x.(a \sqcap x) \sqcup b$ with $a \in \{\bot, \top\}$. Thus, ascending chains of functions have length at most $3 \cdot \#V$.

Let E denote a set of edges and $[.] : E \to \mathbb{F}$ denote an assignment of functions to edges. Then we extend $[.]$ to sequences $w = e_1 \ldots e_n \in E^*$ and sets $M \subseteq E^*$ in the natural way, i.e., by

$$[w] = [e_n] \circ \ldots \circ [e_1] \qquad\qquad [M] = \bigsqcup\{[w] \mid w \in M\}$$

Thus, especially, $[\emptyset] = \lambda x.\bot$ (the least element in $\mathbb{F}$), and $[\{\epsilon\}] = [\epsilon] = I$. Functions $[w]$ and $[M]$ are also called the *effect* of the sequence w and the set M, respectively.

For the rest of this paper we assume that we are given an assignment

$$[e] = f_e = \lambda x.(a_e \sqcap x) \sqcup b_e \in \mathbb{F}$$

to each basic edge e of our input program. Then program analysis tries to compute (approximations to) the following values:

Effects of Procedures: For each procedure p, $\mathsf{Effect}(p) := [\Pi(p)]$ denotes the effect of the set of all same-level execution paths through p;

Reachability: For a start value $d_0 \in \mathbb{D}$, program point v and procedure p, $\mathsf{Reach}(v) := [\Pi_r(v)]\, d_0$ and $\mathsf{Reach}(p) := [\Pi_r(p)]\, d_0$ denote the least upper bounds on all abstract values reaching v along execution paths from main and the least upper bound on all abstract values reaching calls to p, respectively.

The system of these values is called the *Merge-Over-all-Paths* solution (abbreviated: MOP solution) of the analysis problem. Since the respective sets of execution paths are typically infinite, it is not clear whether this solution can be computed effectively. The standard approach proposed in data-flow analysis and abstract interpretation [3, 4, 2] consists in putting up a set $\mathcal{C}$ of constraints on the values we are interested in. The constraints are chosen in such a way that any solution to $\mathcal{C}$ is guaranteed to represent a safe approximation of the values. Quite frequently, however, the least solution of $\mathcal{C}$ equals the MOP solution [8, 13]. Then we speak of *coincidence* of the solutions, meaning that $\mathcal{C}$ *precisely* characterizes the MOP.

In our present application, we are already given a constraint system whose least solution represents the sets of execution paths which are to be evaluated. By inspecting this constraint system, we would naturally try to obtain constraint systems for effect analysis and reachability just by abstracting the lattice of sets of paths with our lattice $\mathbb{F}$. Thus, the ordering "$\subseteq$" of set inclusion on sets of paths is mapped to the ordering on $\mathbb{F}$; set union and concatenation is mapped to least upper bounds and composition of functions. Indeed, this abstraction mapping $[.]$ has the following properties:

Proposition 1. *Let $M_1, M_2 \subseteq E^*$. Then the following holds:*

1. $[M_1 \cup M_2] = [M_1] \sqcup [M_2]$;
2. $[M_1 \cdot M_2] = [M_2] \circ [M_1]$ *if both M_1 and M_2 are non-empty.* □

Proposition 1 suggests a direct translation of the constraint system for the sets of execution paths into a constraint system which we are aiming at. The only two obstacles withstanding a direct translation are (1) an abstract interleaving operator (which for simplicity is denoted by "$\otimes$" as well), and (2) a way how to deal with prefixes. For our abstract lattices, these two problems turn out to have surprisingly simple solutions.

For $f_i = \lambda x.(a_i \sqcap x) \sqcup b_i$, $i = 1, 2$, we define the *interleaving* of f_1 and f_2 by:

$$f_1 \otimes f_2 = \lambda x.(a_1 \sqcap a_2 \sqcap x) \sqcup b_1 \sqcup b_2$$

We have:

Proposition 2. *Let $f_1, f_2, f \in \mathbb{F}$. Then the following holds:*

1. $f_1 \otimes f_2 = f_1 \circ f_2 \sqcup f_2 \circ f_1$;
2. $(f_1 \sqcup f_2) \otimes f = f_1 \otimes f \sqcup f_2 \otimes f$;
3. $[M_1 \otimes M_2] = [M_1] \otimes [M_2]$ *for non-empty subsets $M_1, M_2 \subseteq E^*$.*

For a proof of Proposition 2 see appendix A. Let us now consider the set $\text{pre}(M)$ of prefixes of a non-empty set $M \subseteq E^*$. Then the following holds:

Proposition 3. *Let E_M denote the edges occurring in elements of M where for $e \in E_M$, $[e] = \lambda x.(a_e \sqcap x) \sqcup b_e$. Then*

$$[pre(M)] = \lambda x.x \sqcup B \qquad \text{where} \qquad B = \bigsqcup\{b_e \mid e \in E_M\} \qquad \square$$

Thus, all the intersections with the a_e have disappeared. What only remains is the least upper bound on the values b_e.

5 Effect Analysis

Now we have all prerequisites together to present a constraint system for effect analysis. The least solution of the constraint system defines values $[p]$ for the effect of procedures p together with values $[v]$ for the effects of same-level execution paths reaching program point v.

$$\begin{array}{lll}
[p] \sqsupseteq [r] & r \text{ return point of } p & (1) \\
[s] \sqsupseteq I & s \text{ entry point} & (2) \\
[v] \sqsupseteq f_e \circ [u] & e = (u, v) \text{ basic edge} & (3) \\
[v] \sqsupseteq [p] \circ [u] & e = (u, v) \text{ calls } p & (4) \\
[v] \sqsupseteq ([p_1] \otimes [p_2]) \circ [u] & e = (u, v) \text{ calls } p_1 \,||\, p_2 & (5)
\end{array}$$

Lines (1) through (4) are the lines to determine the effects of procedures as known from inter-procedural analysis of sequential programs. Line (1) says that the effect of procedure p is the effect of what has been accumulated for the return point of p. Line (2) says that accumulation of effects starts at entry points of procedures with the identity function $I = \lambda x.x$. Line (3) says that the contribution of a basic edge $e = (u, v)$ to the value for v is given by the value for u extended by the application of the function f_e associated with this edge. Line (4) says that the contribution of an edge $e = (u, v)$ calling a procedure p is determined analogously with the only difference that the function f_e in line (3) is now replaced with the effect $[p]$ of the called procedure. Also line (5) for a parallel call has the same form. But now, in order to determine the combined effect of the parallely executed procedures p_1 and p_2, we rely on the interleaving operator "$\otimes$". This constraint system for effect analysis is the direct abstraction of the corresponding constraint system for same-level reaching paths from section 3. Therefore, we obtain (by distributivity of all involved operators):

Theorem 1. *The least solution of the effect constraint system precisely describes the effect of procedures, i.e.,*

$$\mathsf{Effect}(p) = [p] \qquad \textit{and} \qquad \mathsf{Effect}(v) = [v]$$

for every procedure p and program point v. These values can be computed in time $\mathcal{O}(h \cdot n)$ where n is the size of the program. □

6 A Constraint System for Reachability

As for effect analysis, we could mimic the least fixpoint definition of the sets of reaching execution paths through a corresponding constraint system over $\mathbb{F}$. Observe, however, that our defining constraint system for reaching execution paths in section 3 has quadratic size. Clearly, we would like to improve on this, and indeed this is possible – even without sacrificing precision.

Instead of accumulating effects in a topdown fashion as was necessary in the precise definition of reaching execution paths, we prefer a bottom-up accumulation – a strategy which is commonly used in inter-procedural analysis of sequential programs. There, accumulation directly starts at the main program and then successively proceeds to called procedures.

For each program point v, let $B(v)$ denote the least upper bound of all b_e, for all basic edges e possibly executed in parallel with v. This value is also called *possible interference* of v. Formally, these values are determined through the least solution of the following constraint system:

$$\begin{array}{lll}
\sigma(p) & \sqsupseteq b_e & e \text{ basic edge in procedure } p \\
\sigma(p) & \sqsupseteq \sigma(q) & \text{procedure } p \text{ calls } q \text{ or } q \,||\, \_ \text{ or } \_ \,||\, q \\
B(v) & \sqsupseteq B(p) & v \text{ program point in } p \\
B(p) & \sqsupseteq B(u) & (u, \_) \text{ calls procedure } p \\
B(q_i) & \sqsupseteq \sigma(q_{3-i}) \sqcup B(u) & (u, \_) \text{ calls } q_1 \,||\, q_2
\end{array}$$

We used auxiliary values $\sigma(p)$, p a procedure, to calculate the least upper bound on b_e for all basic edges possibly executed during evaluation of p. The whole system for computing the values $\sigma(p)$, $B(p)$ and $B(v)$ is of linear size and uses "$\sqcup$" as only operation in right-hand sides. Such kind of problems are also known as "pure merge problems" and can be solved even in linear time.
We will now construct a constraint system as for inter-procedural reachability analysis of sequential programs, but for each program point additionally take its possible interference into account. Thus, we consider the values $[\![v]\!]$, v a program point, $[\![p]\!]$, p a procedure, which are determined as the least solution of the following constraint system:

$$\begin{array}{lll}[\![\mathsf{main}]\!] \sqsupseteq d_0 & & (1)\\ [\![v]\!] \sqsupseteq B(v) & \text{and} & (2)\\ [\![v]\!] \sqsupseteq [v][\![p]\!] & v \text{ program point in procedure } p & (3)\\ [\![p]\!] \sqsupseteq [\![u]\!] & e = (u, \_) \text{ calls } p \text{ or } p \,||\, \_ \text{ or } \_ \,||\, p & (4)\end{array}$$

Only line (2) makes the difference to a corresponding constraint system for reachability in sequential programs. The intuition behind the constraint system is as follows. Line (1) says that initially the value reaching main should subsume the initial value d_0. Line (2) says that the value reaching program point v should subsume its possible interference. Line (3) says that when v is a program point of procedure p, then the reaching value should also subsume the intra-procedural effect of v applied to the value reaching p. Line (4) finally says that the value reaching a procedure should subsume the value of every program point where such a call (possibly in parallel to another call) is possible.
This constraint system differs considerably from the constraint system for the sets of reaching execution paths. Nonetheless, we are able to prove:

Theorem 2. *The above constraint system computes precise reachability information as well, i.e.,*

$$\mathsf{Reach}(p) = [\![p]\!] \qquad \textit{and} \qquad \mathsf{Reach}(v) = [\![v]\!]$$

for all program points v and procedures p. These values can be computed in time $\mathcal{O}(h \cdot n)$ where n is the size of the program.

For a proof see appendix B. Theorem 2 implies that programs with procedures and parallelism are not harder to analyze than programs with procedures but without parallelism!

7 Extensions

In this section, we discuss issues which are important for the practical applicability of the presented results. We do not claim that this section contains any new ideas or constructions. Rather we want to emphasize that the constructions known from the inter-procedural analysis of sequential programs can be extended to parallel programs in a straight-forward way.

7.1 Non-reachable Program Points

So far, we assumed that every program point is reachable by at least one execution path. In order to show that this assumption is not vital, let $\mathcal{P}$ and $\mathcal{R}$ denote the sets of possibly terminating procedures and reachable program points, respectively. In order to compute these sets, we instantiate our generic analysis with $\mathbb{D} = \{0 \sqsubset 1\}$ where for each basic edge e, the function $[e] = f_e$ is given by $f_e = I = \lambda x.x$, and the initial value d_0 equals 1. The only functions from $\mathbb{D} \to \mathbb{D}$ occurring during the analysis are $\lambda x.\bot$ and I. Both functions are *strict*, i.e., map $\bot$ to $\bot$. Therefore, we obtain:

Proposition 4. *For every procedure p and program point v, the following holds:*

1. $[v] = I$ *iff* $\Pi(v) \neq \emptyset$ *and* $[p] = I$ *iff* $\Pi(p) \neq \emptyset$;
2. $[\![v]\!] = 1$ *iff* $\Pi_r(v) \neq \emptyset$ *and* $[\![p]\!] = 1$ *iff* $\Pi_r(p) \neq \emptyset$.

In particular, $p \in \mathcal{P}$ iff $[p] = I$, and $v \in \mathcal{R}$ iff $[\![v]\!] = 1$. □

We conclude that the sets $\mathcal{P}$ and $\mathcal{R}$ can be computed in linear time.

A non-reachable program point should not influence any other program point. Therefore, we modify the given cfg by removing all edges starting in program points not in $\mathcal{R}$. By this edge removal, the sets of reaching execution paths have not changed. Let us call the resulting cfg *normalized.* Then we obtain:

Theorem 3. *Assume the cfg is normalized. Then for every program point v and procedure p,*

1. $\mathsf{Effect}(v) = [v]$ *and* $\mathsf{Effect}(p) = [p]$;
2. $\mathsf{Reach}(v) = [\![v]\!]$ *and* $\mathsf{Reach}(p) = [\![p]\!]$. □

We conclude that, after the preprocessing step of normalization, our constraint systems will compute a safe approximation which is precise.

Practical Remark: Normalization of the cfg may remove edges and thus some constraints from the constraint systems of the analysis. Therefore, omitting normalization may result in a less precise, but still safe analysis.

7.2 Backward Analysis

What we discussed so far, is called *forward analysis.* Examples of forward analysis problems are reaching definitions, available expressions or simple constant propagation. Other important analyses, however, determine the value at a program point v w.r.t. the *possible future* of v, i.e., the set of reverses of execution paths possibly following a visit of v. Examples are live variables or very busy expressions. Such analyses are called *backward analyses.* In case that every forward reachable program point is also backward reachable, i.e., lies on an execution path from the start point to the return point of `main`, we can reduce backward analysis to forward analysis – simply by normalizing the cfg followed by a reversal of edge orientations and an exchange of entry and return points of procedures.

7.3 Local and Global State

Consider an edge $e = (u, v)$ in the cfg which calls a terminating procedure p (the treatment of a terminating parallel call to two procedures p_1 and p_2 is completely analogous). So far, the complete information at program point u is passed to the entry point of p. Indeed, this is adequate when analyzing *global* properties like availability of expressions which depend on global variables only. It is not (immediately) applicable in presence of *local* variables which are visible to the caller but should be hidden from the callee p, meaning that they should survive the call unchanged [8, 13].

To make things precise, let us assume that $\mathbb{D} = \mathbb{D}_l \times \mathbb{D}_g$ where $\mathbb{D}_l$ and $\mathbb{D}_g$ describe local and global properties, respectively. Let us further assume that the global part of the current state is passed as a parameter to p, and also returned as the result of the call, whereas the local part of the program point before the call is by-passed the call using some transformer $\beta_e : \mathbb{D}_l \to \mathbb{D}_l$. Recall that every $f \in \mathbb{F}$ is of the form $f = \lambda x.(x \sqcap a) \sqcup b$ with $a, b \in \mathbb{D}$. Since $\mathbb{D}$ is a Cartesian product, this implies that $f = f_l \times f_g$ where $f_l : \mathbb{D}_l \to \mathbb{D}_l$ and $f_g : \mathbb{D}_g \to \mathbb{D}_g$ independently operate on the local states and global states, respectively.

Therefore, we can separate the analysis into two phases.

The first phase considers just global values from $\mathbb{D}_g$. No local state need to be preserved during the call, and we use the original call edge.

The second phase then is purely intra-procedural and deals with the lattice $\mathbb{D}_l$. But now, since the call at edge e has no effect onto the local state, we simply change e into a basic edge with $[e] = \beta_e$.

8 Conclusion and Perspectives

We have shown how to extend the intra-procedural method of [11] to uniformly and efficiently capture inter-procedural bitvector analyses of fork/join parallel programs. Our method, which comprises analysis problems like available expressions, live variables or simple constant propagation, passes the test for practicality, as it 'behaves' as the widely accepted algorithms for sequential inter-procedural program analysis. Moreover, even though precision can only be proved for fork/join parallelism, our algorithm may also be used for computing safe approximations for languages with arbitrary synchronization statements. Finally, due to its structural similarity to the sequential case, it can easily be integrated in program analysis environments like e.g. MetaFrame or PAG, which already contain the necessary fixpoint machinery.

As a next step, we plan a closer comparison with the automata theoretic approach of Esparza and Podelski. The considered program structures are obviously similar, however, the range of possible analyses may be different. As shown in [12], the automata theoretic approach is able to capture the model checking problem for all of the linear time temporal logic *EF*. It would be interesting to see whether it is possible to adopt our technique to covering this logic as well, or whether the automata theoretic approach, which is significantly more complex already for the analysis problem considered here, is inherently more powerful.

References

[1] Martin Alt and Florian Martin. Generation of Efficient Interprocedural Analyzers with PAG. In *Proceedings of 2nd Static Analysis Symposium (SAS)*, pages 33–50. LNCS 983, Springer Verlag, 1995.

[2] Patrick Cousot. Semantic Foundations of Program Analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[3] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of 4th ACM Symposium on Principles of Programming Languages (POPL)*, pages 238–252. ACM Press, 1977.

[4] Patrick Cousot and Radhia Cousot. Static Determination of Dynamic Properties of Recursive Programs. In E.J. Neuhold, editor, *Formal Descriptions of Programming Concepts*, pages 237–277. North-Holland Publishing Company, 1977.

[5] J. Esparza and J. Knoop. An Automata-theoretic Approach to Interprocedural Data-flow Analysis. In *FoSSaCS '99*, volume 1578 of *Lecture Notes in Computer Science (LNCS)*, pages 14–30. Springer-Verlag, 1999.

[6] J. Esparza and A. Podelski. Efficient Algorithms for pre* and post* on Interprocedural Parallel Flow Graphs. In *ACM International Conference on Princples of Programming Languages (POPL)*, 2000. To appear.

[7] M.S. Hecht. *Flow Analysis of Computer Programs.* The Computer Science Library. North-Holland, New York, 1977.

[8] J.Knoop and B.Steffen. The Interprocedural Coincidence Theorem. In *4th International Conference on Compiler Construction (CC'92)*, volume 641 of *Lecture Notes in Computer Science (LNCS)*, pages 125–140. Springer-Verlag, 1992.

[9] J. Knoop. Parallel Constant Propagation. In *4th European Conference on Parallel Processing (Euro-Par)*, volume 1470 of *Lecture Notes in Computer Science (LNCS)*, pages 445–455. Springer-Verlag, 1998.

[10] J. Knoop, O. Rüthing, and B. Steffen. Towards a Tool Kit for the Automatic Generation of Interprocedural Data Flow Analyses. *Journal of Programming Languages*, 4(4):211–246, December 1996. Chapman & Hall, London (UK).

[11] J. Knoop, B. Steffen, and J. Vollmer. Parallelism for Free: Efficient and Optimal Bitvector Analyses for Parallel Programs. *ACM Transactions on Programming Languages and Systems*, 18(3):268–299, 1996.

[12] D. Lugiez and P. Schnoebelen. The Regular Viewpoint on PA-Processes. In *9th International Conference on Concurrency (CONCUR)*, volume 1466 of *Lecture Note In Computer Science (LNCS)*, pages 50–66. Springer-Verlag, 1998.

[13] H. Seidl and C. Fecht. Interprocedural Analyses: A Comparison. *Journal of Logic Programming (JLP)*, 43(2):123–156, 2000.

A Proof of Proposition 2

We only prove statement (3). Let M_1, M_2 be non-empty subsets of E^*. By statement (1), we have

$$\begin{aligned} [M_1] \otimes [M_2] &= [M_1] \circ [M_2] \sqcup [M_2] \circ [M_1] \\ &= [M_2 \cdot M_1 \cup M_1 \cdot M_2] \qquad \sqsubseteq [M_1 \otimes M_2] \end{aligned}$$

Therefore, it remains to prove the reverse inequality. For that consider $w = e_1 \dots e_m \in M_1 \otimes M_2$ where for disjoint index sets I_1, I_2 with $I_1 \cup I_2 = \{1, \dots, m\}$, $w_i = w|_{I_i} \in M_i$. We claim:

$$[w] \sqsubseteq [w_1] \circ [w_2] \sqcup [w_2] \circ [w_1]$$

Clearly, this claim implies the statement (3) of our proposition. In order to prove the claim, let $[e_i] = \lambda x.(a_i \sqcap x) \sqcup b_i$, $i = 1, \ldots, m$, $[w_j] = \lambda x.(A_j \sqcap x) \sqcup B_j$, $j = 1, 2$, and $[w] = \lambda x.(A \sqcap x) \sqcup B$. Then by definition,

$$A = a_1 \sqcap \ldots \sqcap a_m = A_1 \sqcap A_2$$

Now consider value B. By definition,

$$B = \bigsqcup_{k=1}^{m} (b_k \sqcap a_{k+1} \sqcap \ldots \sqcap a_m)$$

We will show that for every k,

$$b_k \sqcap a_{k+1} \sqcap \ldots \sqcap a_m \sqsubseteq B_1 \sqcup B_2$$

W.l.o.g. assume that $k \in I_1$ (the case where $k \in I_2$ is completely analogous) and let $\{j_1, \ldots, j_r\} = \{j \in I_1 \mid j > k\}$. Then

$$b_k \sqcap a_{k+1} \sqcap \ldots \sqcap a_m \sqsubseteq b_k \sqcap a_{j_1} \sqcap \ldots \sqcap a_{j_r} \sqsubseteq B_1$$

which implies the assertion. □

B Proof of Theorem 2

Let us start with the following simple but useful observation:

Proposition 5. *For every $f \in \mathbb{F}$, $b \in \mathbb{D}$ and $\Delta = \lambda x.x \sqcup b$,*

$$f \otimes \Delta = \Delta \circ f$$ □

Next, we reformulate the constraint system for reachability as follows. We introduce the new values $[\![v]\!]'$, v a program point, and $[\![p]\!]'$, p a procedure, which collect the least upper bounds of *directly* reaching values by ignoring possible interleavings with execution paths possibly executed in parallel to v (or p). These values are determined as the least solution of the following constraint system:

$$\begin{array}{lll l}
[\![\mathsf{main}]\!]' \sqsupseteq d_0 & & & (1)\\
[\![v]\!]' \sqsupseteq [v] \circ [\![p]\!]' & & v \text{ program point in procedure } p & (2)\\
[\![p]\!]' \sqsupseteq [\![u]\!]' & & e = (u, \_) \text{ calls } p \text{ or } p \,||\, \_ \text{ or } \_ \,||\, p & (3)
\end{array}$$

By standard fixpoint induction we find:

Proposition 6. *For all program points v and procedures p,*

$$[\![v]\!] = [\![v]\!]' \sqcup B(v) \qquad \textit{and} \qquad [\![p]\!] = [\![p]\!]' \sqcup B(p)$$ □

In order to understand the "nature" of the values $B(v)$, we consider the sets $P(v)$ of edges possibly executed in parallel with program points v. They are determined through the least solution of the following constraint system:

$$\begin{array}{lll}
E(p) & \supseteq \{e\} & e \text{ basic edge in procedure } p\\
E(p) & \supseteq E(q) & \text{procedure } p \text{ calls } q \text{ or } q \,||\, \_ \text{ or } \_ \,||\, q\\
P(v) & \supseteq P(p) & v \text{ program point in } p\\
P(p) & \supseteq P(u) & (u, \_) \text{ calls procedure } p\\
P(q_i) & \supseteq E(q_{3-i}) \cup P(u) & (u, \_) \text{ calls } q_1 \,||\, q_2
\end{array}$$

By comparison of this constraint system with the definition of the values $\sigma(p)$ and $B(v), B(p)$ in section 6, we obtain:

Proposition 7. *For every procedure p and program point v,*

1. $\lambda x.x \sqcup \sigma(p) = I \sqcup [E(p)]$;
2. $\lambda x.x \sqcup B(p) = I \sqcup [P(p)]$ *and* $\lambda x.x \sqcup B(v) = I \sqcup [P(v)]$. □

Moreover, we have:

Proposition 8. *For every procedure p,*

$$[pre(\Pi(p))] = I \sqcup \bigsqcup[E(p)] = \lambda x.x \sqcup \sigma(p) \qquad \square$$

In order to simplify the proof of theorem 2, let us assume that all calls are parallel calls $q_1 \,||\, q_2$. This assumption does not incur a restriction, since an ordinary call to a procedure p can easily be simulated by a call to $p \,||\, q_0$ where q_0 is a procedure with just a single program point and no edges at all. Furthermore, it suffices to prove the assertion of the theorem just for program points v (the assertion for procedures then is an immediate consequence). We want to prove that for every program point v, the value $[\![v]\!]$ is a safe approximation of the value $\mathsf{Reach}(v)$, i.e., $[\![v]\!] \sqsupseteq \mathsf{Reach}(v)$. By definition,

$$\mathsf{Reach}(v) = [\Pi_r(v)]\, d_0 = [\Pi(v, \mathsf{main})]\, d_0$$

Therefore, let $w \in \Pi(v, \mathsf{main})$. Then there are program points $u_0, \ldots, u_m$, execution paths $w_0, \ldots, w_m$ together with execution paths w'_i, procedures $q_1^{(i)}, q_2^{(i)}$ and indices $j(i) \in \{1, 2\}$ for $i = 1, \ldots, m$ such that:

- $u_m = v$;
- $w_i \in \Pi(u_i)$ for $i = 0, \ldots, m$;
- there are calls $(u_{i-1}, \_)$ to $q_1^{(i)} \,||\, q_2^{(i)}$;
- u_0 is a program point in main and for $i > 0$, u_i is a program point in $q_{j(i)}^{(i)}$;
- $w'_i \in \text{pre}(\Pi(q_{3-j(i)}^{(i)}))$ for $i = 1, \ldots, m$;
- $w \in \{w_0\} \cdot (\{w'_1\} \otimes (\{w_1\} \cdot (\ldots \{w'_{m-1}\} \otimes (\{w_{m-1}\} \cdot (\{w'_m\} \otimes \{w_m\}))\ldots)))$.

Let $\Delta = \lambda x.x \sqcup P(v)$. Then by proposition 8,

$$[w'_i] \sqsubseteq [\text{pre}(\Pi(q_{3-j(i)}^{(i)}))] = I \sqcup [E(q_{3-j(i)}^{(i)})] \sqsubseteq I \sqcup [P(v)] = \Delta$$

for all $i = 1, \ldots, m$. Therefore by proposition 5,

$$\begin{array}{rcl} [w] & \sqsubseteq & (((\ldots(([w_m] \otimes [w'_m]) \circ [w_{m-1}]) \otimes [w'_{m-1}] \ldots) \circ [w_1]) \otimes [w'_1]) \circ [w_0] \\ & \sqsubseteq & (((\ldots(([w_m] \otimes \Delta) \circ [w_{m-1}]) \otimes \Delta \ldots) \circ [w_1]) \otimes \Delta) \circ [w_0] \\ & = & \Delta \circ (\ldots(([w_m] \otimes \Delta) \circ [w_{m-1}]) \otimes \Delta \ldots) \circ [w_1] \circ [w_0] \\ & \ldots & \\ & = & \Delta \circ [w_m] \circ [w_{m-1}] \circ \ldots \circ [w_0] \end{array}$$

Since $([w_m] \circ \ldots \circ [w_0])\, d_0 \sqsubseteq [\![v]\!]'$, we conclude that

$$[w]\, d_0 \sqsubseteq \Delta\, [\![v]\!]' = [\![v]\!]' \sqcup B(v) = [\![v]\!]$$

which we wanted to prove.
It remains to prove the reverse inequality, i.e., that (1) $[\![v]\!]' \sqsubseteq \mathsf{Reach}(v)$ and (2) $B(v) \sqsubseteq \mathsf{Reach}(v)$.
Let us first consider inequality (1). The value $[\![v]\!]'$ is the least upper bound on values $[w]\,d_0$ such that there exit program points $u_0, \ldots, u_m$, execution paths $w_0, \ldots, w_m$ together with procedures $q_1^{(i)}, q_2^{(i)}$ and indices $j(i) \in \{1, 2\}$ for $i = 1, \ldots, m$ such that:

- $u_m = v$;
- $w_i \in \Pi(u_i)$ for $i = 0, \ldots, m$;
- there are calls $(u_{i-1}, \_)$ to $q_1^{(i)} \,||\, q_2^{(i)}$;
- u_0 is a program point in main and for $i > 0$, u_i is a program point in $q_{j(i)}^{(i)}$;
- $w = w_0 \ldots w_m$.

By induction on $r = m - i$ (from $r = 0$ to $r = m - 1$), we find that for $i > 0$,

$$w_i \ldots w_m \in \Pi(v, q_{j(i)}^{(i)})$$

and for $i = 0$,

$$w = w_0 \ldots w_m \in \Pi(v, \mathsf{main}) = \Pi_r(v)$$

Therefore,

$$[w]\,d_0 \sqsubseteq [\Pi_r(v)]\,d_0 = \mathsf{Reach}(v)$$

which we wanted to prove.
Now let us consider inequality (2). By proposition 7, $\lambda x.x \sqcup B(v) = I \sqcup [P(v)]$. Therefore, it suffices to prove for each edge $e \in P(v)$, that $b_e \sqsubseteq \mathsf{Reach}(v)$.
Since $e \in P(v)$, there exist program points $u_0, \ldots, u_m$, execution paths $w_0, \ldots, w_m$ together with procedures $q_1^{(i)}, q_2^{(i)}$, indices $j(i) \in \{1, 2\}$ for $i = 1, \ldots, m$, an index $k \in \{1, \ldots, m\}$ and one execution path w' such that

- $u_m = v$;
- $w_i \in \Pi(u_i)$ for $i = 0, \ldots, m$;
- there are calls $(u_{i-1}, \_)$ to $q_1^{(i)} \,||\, q_2^{(i)}$;
- u_0 is a program point in main and for $i > 0$, u_i is a program point in $q_{j(i)}^{(i)}$;
- $w'e \in \mathrm{pre}(\Pi(q_{3-j(k)}^{(k)})$.

As above, we conclude that $w_k \ldots w_m \in \Pi(v, q_{j(k)}^{(k)})$. By definition, then also

$$w_{k-1} w_k \ldots w_m w' e \in \Pi(v, q_{j(k-1)}^{(k-1)})$$

(where in case $k = 1$, we let $q_{j(0)}^{(0)} = \mathsf{main}$) and therefore also

$$w_0 \ldots w_{k-1} w_k \ldots w_m w' e \in \Pi(v, \mathsf{main}) = \Pi_r(v)$$

We conclude that

$$b_e \sqsubseteq b_e \sqcup (a_e \sqcap ([w_0 \ldots w_m w']\ d_0)) = [w_0 \ldots w_m w' e]\, d_0 \sqsubseteq [\Pi_r(v)]\, d_0 = \mathsf{Reach}(v)$$

which completes the proof. □

Alias Types *

Frederick Smith, David Walker, and Greg Morrisett

Cornell University

Abstract. Linear type systems allow destructive operations such as object deallocation and imperative updates of functional data structures. These operations and others, such as the ability to reuse memory at different types, are essential in low-level typed languages. However, traditional linear type systems are too restrictive for use in low-level code where it is necessary to exploit pointer aliasing. We present a new typed language that allows functions to specify the shape of the store that they expect and to track the flow of pointers through a computation. Our type system is expressive enough to represent pointer aliasing and yet safely permit destructive operations.

1 Introduction

Linear type systems [26, 25] give programmers explicit control over memory resources. The critical invariant of a linear type system is that every linear value is used exactly once. After its single use, a linear value is dead and the system can immediately reclaim its space or reuse it to store another value. Although this single-use invariant enables compile-time garbage collection and imperative updates to functional data structures, it also limits the use of linear values. For example, x is used twice in the following expression: $\texttt{let}\ x = \langle 1, 2\rangle\ \texttt{in let}\ y = fst(x)\ \texttt{in let}\ z = snd(x)\ \texttt{in}\ y + z$. Therefore, x cannot be given a linear type, and consequently, cannot be deallocated early.

Several authors [26, 9, 3] have extended pure linear type systems to allow greater flexibility. However, most of these efforts have focused on high-level user programming languages and as a result, they have emphasized simple typing rules that programmers can understand and/or typing rules that admit effective type inference techniques. These issues are less important for low-level typed languages designed as compiler intermediate languages [22, 18] or as secure mobile code platforms, such as the Java Virtual Machine [10], Proof-Carrying Code (PCC) [13] or Typed Assembly Language (TAL) [12]. These languages are designed for machine, not human, consumption. On the other hand, because systems such as PCC and TAL make every machine operation explicit and verify that each is safe, the implementation of these systems requires new type-theoretic mechanisms to make efficient use of computer resources.

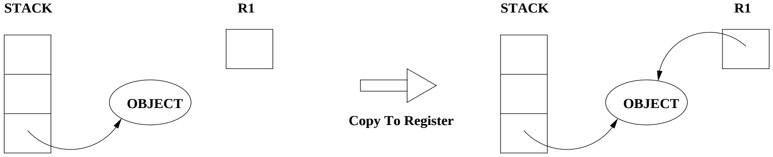
* This material is based on work supported in part by the AFOSR grant F49620-97-1-0013 and the National Science Foundation under Grant No. EIA 97-03470. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not reflect the views of these agencies.

G. Smolka (Ed.): ESOP/ETAPS 2000, LNCS 1782, pp. 366–381, 2000.

In existing high-level typed languages, every location is stamped with a single type for the lifetime of the program. Failing to maintain this invariant has resulted in unsound type systems or misfeatures (witness the interaction between parametric polymorphism and references in ML [23, 27]). In low-level languages that aim to expose the resources of the underlying machine, this invariant is untenable. For instance, because machines contain a limited number of registers, each register cannot be stamped with a single type. Also, when two stack-allocated objects have disjoint lifetimes, compilers naturally reuse the stack space, even when the two objects have different types. Finally, in a low-level language exposing initialization, even the simplest objects change type. For example, a pair x of type $\langle int, int\rangle$ may be created as follows:

$$\begin{array}{ll}
\texttt{malloc}\, x, 2\ ; & \texttt{(*}\ x \text{ has type } \langle \mathit{junk}, \mathit{junk}\rangle\ \texttt{*)} \\
x[1]{:=}1\ ; & \texttt{(*}\ x \text{ has type } \langle \mathit{int}, \mathit{junk}\rangle\ \texttt{*)} \\
x[2]{:=}2\ ; & \texttt{(*}\ x \text{ has type } \langle \mathit{int}, \mathit{int}\rangle\ \texttt{*)} \\
\vdots &
\end{array}$$

At each step in this computation, the storage bound to x takes on a different type ranging from nonsense (indicated by the type *junk*) to a fully initialized pair of integers. In this simple example, there are no aliases of the pair and therefore we might be able to use linear types to verify that the code is safe. However, in a more complex example, a compiler might generate code to compute the initial values of the tuple fields between allocation and the initializing assignments. During the computation, a register allocator may be forced to move the uninitialized or partially initialized value x between stack slots and registers, creating aliases:



If x is a linear value, one of the pointers shown above would have to be "invalidated" in some way after each move. Unfortunately, assuming the pointer on the stack is invalidated, future register pressure may force x to be physically copied back onto the stack. Although this additional copy is unnecessary because the register allocator can easily remember that a pointer to the data structure remains on the stack, the limitations of a pure linear type system require it.

Pointer aliasing and data sharing also occur naturally in other data structures introduced by a compiler. For example, compilers often use a top-of-stack pointer and a frame pointer, both of which point to the same data structure. Compiling a language like Pascal using displays [1] generalizes this problem to having an arbitrary (but statically known) number of pointers into the same data structure. In each of these examples, a flexible type system will allow aliasing but ensure that no inconsistencies arise. Type systems for low-level languages, therefore, should support values whose types change even when those values are aliased.

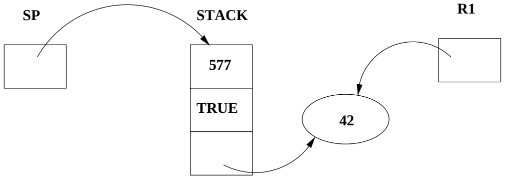
We have devised a new type system that uses linear reasoning to allow memory reuse at different types, object initialization, safe deallocation, and tracking of sharing in data structures. This paper formalizes the type system and provides a theoretical foundation for safely integrating operations that depend upon pointer aliasing with type systems that include polymorphism and higher-order functions.

We have extended the TAL implementation with the features described in this paper.[1] It was quite straightforward to augment the existing F^{ω}-based type system because many of the basic mechanisms, including polymorphism and singleton types, were already present in the type constructor language. Popcorn, an optimizing compiler for a safe C-like language, generates code for the new TAL type system and uses the alias tracking features of our type system.

The Popcorn compiler and TAL implementation demonstrate that the ideas presented in this paper can be integrated with a practical and complete programming language. However, for the sake of clarity, we only present a small fragment of our type system and, rather than formalizing it in the context of TAL, we present our ideas in terms of a more familiar lambda calculus. Section 2 gives an informal overview of how to use *aliasing constraints*, a notion which extends conventional linear type systems, to admit destructive operations such as object deallocation in the presence of aliasing. Section 3 describes the core language formally, with emphasis on the rules for manipulating linear aliasing constraints. Section 4 extends the language with non-linear aliasing constraints. Finally, Section 5 discusses future and related work.

2 Informal Overview

The main feature of our new type system is a collection of *aliasing constraints*. Aliasing constraints describe the shape of the store and every function uses them to specify the store that it expects. If the current store does not conform to the constraints specified, then the type system ensures that the function cannot be called. To illustrate how our constraints abstract a concrete store, we will consider the following example:

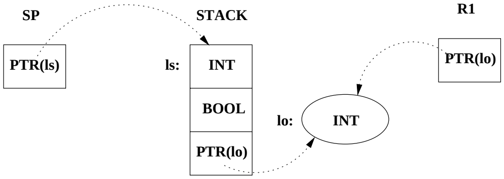


Here, *sp* is a pointer to a stack frame, which has been allocated on the heap (as might be done in the SML/NJ compiler [2], for instance). This frame contains a pointer to a second object, which is also pointed to by register r_1.

In our program model, every heap-allocated object occupies a particular memory location. For example, the stack frame might occupy location ℓ_s and the

[1] See http://www.cs.cornell.edu/talc for the latest software release.

second object might occupy location ℓ_o. In order to track the flow of pointers to these locations accurately, we reflect locations into the type system: A pointer to a location ℓ is given the singleton type *ptr*(ℓ). Each singleton type contains exactly one value (the pointer in question). This property allows the type system to reason about pointers in a very fine-grained way. In fact, it allows us to represent the graph structure of our example store precisely:



We represent this picture in our formal syntax by declaring the program variable *sp* to have type $ptr(\ell_s)$ and r_1 to have type $ptr(\ell_o)$. The store itself is described by the constraints $\{\ell_s \mapsto \langle int, bool, ptr(\ell_o)\rangle\} \oplus \{\ell_o \mapsto \langle int\rangle\}$, where the type $\langle\tau_1, \ldots, \tau_n\rangle$ denotes a memory block containing values with types τ_1 through τ_n.

Constraints of the form $\{\ell \mapsto \tau\}$ are a reasonable starting point for an abstraction of the store. However, they are actually *too precise* to be useful for general-purpose programs. Consider, for example, the simple function *deref*, which retrieves an integer from a reference cell. There are two immediate problems if we demand that code call *deref* when the store has a shape described by $\{\ell \mapsto \langle int\rangle\}$. First, *deref* can only be used to derefence the location ℓ, and not, for example, the locations ℓ' or ℓ''. This problem is easily solved by adding *location polymorphism.* The exact name of a location is usually unimportant; we need only establish a dependence between pointer type and constraint. Hence we could specify that *deref* requires a store $\{\rho \mapsto \langle int\rangle\}$ where ρ is a location variable instead of some specific location ℓ. Second, the constraint $\{\ell \mapsto \langle int\rangle\}$ specifies a store with exactly one location ℓ although we may want to dereference a single integer reference amongst a sea of other heap-allocated objects. Since *deref* does not use or modify any of these other references, we should be able to abstract away the size and shape of the rest of the store. We accomplish this task using *store polymorphism.* An appropriate constraint for the function *deref* is $\epsilon \oplus \{\rho \mapsto \langle int\rangle\}$ where ϵ is a constraint variable that may instantiated with any other constraint.

The third main feature of our constraint language is the capability to distinguish between linear constraints $\{\rho \mapsto \tau\}$ and non-linear constraints $\{\rho \mapsto \tau\}^\omega$. Linear constraints come with the additional guarantee that the location on the left-hand side of the constraint (ρ) is not aliased by any other location (ρ'). This invariant is maintained despite the presence of location polymorphism and store polymorphism. Intuitively, because ρ is unaliased, we can safely deallocate its memory or change the types of the values stored there. The key property that makes our system more expressive than traditional linear systems is that although the aliasing constraints may be linear, the pointer values that flow through a computation are not. Hence, there is no *direct* restriction on the copying and reuse of pointers.

The following example illustrates how the type system uses aliasing constraints and singleton types to track the evolution of the store across a series of instructions that allocate, initialize, and then deallocate storage. In this example, the instruction `malloc` x, ρ, n allocates n words of storage. The new storage is allocated at a fresh location ℓ in the heap and ℓ is substituted for ρ in the remaining instructions. A pointer to ℓ is substitued for x. Both ρ and x are considered bound by this instruction. The `free` instruction deallocates storage. Deallocated storage has type *junk* and the type system prevents any future use of that space.

	Instructions	Constraints (Initially the constraints ϵ)	
1.	`malloc` $sp, \rho_1, 2$;	$\epsilon \oplus \{\rho_1 \mapsto \langle \mathit{junk}, \mathit{junk}\rangle\}$	$sp : ptr(\rho_1)$
2.	$sp[1]{:=}1$;	$\epsilon \oplus \{\rho_1 \mapsto \langle \mathit{int}, \mathit{junk}\rangle\}$	
3.	`malloc` $r_1, \rho_2, 1$;	$\epsilon \oplus \{\rho_1 \mapsto \langle \mathit{int}, \mathit{junk}\rangle, \rho_2 \mapsto \langle \mathit{junk}\rangle\}$	$r_1 : ptr(\rho_2)$
4.	$sp[2]{:=}r_1$;	$\epsilon \oplus \{\rho_1 \mapsto \langle \mathit{int}, ptr(\rho_2)\rangle, \rho_2 \mapsto \langle \mathit{junk}\rangle\}$	
5.	$r_1[1]{:=}2$;	$\epsilon \oplus \{\rho_1 \mapsto \langle \mathit{int}, ptr(\rho_2)\rangle, \rho_2 \mapsto \langle \mathit{int}\rangle\}$	
6.	`free` r_1;	$\epsilon \oplus \{\rho_1 \mapsto \langle \mathit{int}, ptr(\rho_2)\rangle, \rho_2 \mapsto \mathit{junk}\}$	
7.	`free` sp;	$\epsilon \oplus \{\rho_1 \mapsto \mathit{junk}, \rho_2 \mapsto \mathit{junk}\}$	

Again, we can intuitively think of *sp* as the stack pointer and r_1 as a register that holds an alias of an object on the stack. Notice that on line 5, the initialization of r_1 updates the type of the memory at location ρ_2. This has the effect of simultaneously updating the type of r_1 and of $sp[1]$. Both of these paths are similarly affected when r_1 is freed in the next instruction. Despite the presence of the dangling pointer at $sp[1]$, the type system will not allow that pointer to be derefenced.

By using singleton types to accurately track pointers, and aliasing constraints to model the shape of the store, our type system can represent sharing and simultaneously ensure safety in the presence of destructive operations.

3 The Language of Locations

This section describes our new type-safe "language of locations" formally. The syntax for the language appears in Figure 1.

3.1 Values, Instructions, and Programs

A program is a pair of a store (S) and a list of instructions (ι). The store maps locations (ℓ) to values (v). Normally, the values held in the store are memory blocks ($\langle \tau_1, \ldots, \tau_n \rangle$), but after the memory at a location has been deallocated, that location will point to the unusable value `junk`. Other values include integer constants (i), variables (x or f), and, of course, pointers ($\mathtt{ptr}(\ell)$).

Figure 2 formally defines the operational semantics of the language.[2] The main instructions of interest manipulate memory blocks. The instruction

[2] Here and elsewhere, the notation $X[c_1, \ldots, c_n / x_1, \ldots, x_n]$ denotes capture-avoiding substitution of $c_1, \ldots, c_n$ for variables $x_1, \ldots, x_n$ in X.

$\ell \in \text{Locations} \qquad \rho \in \text{LocationVar} \qquad \epsilon \in \text{ConstraintVar} \qquad x, f \in \text{ValueVar}$

$$\begin{array}{lll}
\textit{locations} & \eta ::= & \ell \mid \rho \\
\textit{constraints} & C ::= & \emptyset \mid \epsilon \mid \{\eta \mapsto \tau\} \mid C_1 \oplus C_2 \\
\textit{types} & \tau ::= & \textit{int} \mid \textit{junk} \mid \textit{ptr}(\eta) \mid \langle \tau_1, \ldots, \tau_n \rangle \mid \forall[\Delta; C].(\tau_1, \ldots, \tau_n) \rightarrow 0 \\
\textit{value ctxts} & \Gamma ::= & \cdot \mid \Gamma, x{:}\tau \\
\textit{type ctxts} & \Delta ::= & \cdot \mid \Delta, \rho \mid \Delta, \epsilon \\
\textit{values} & v ::= & x \mid i \mid \texttt{junk} \mid \texttt{ptr}(\ell) \mid \langle v_1, \ldots, v_n \rangle \mid \texttt{fix}\, f[\Delta; C; \Gamma].\iota \mid v[\eta] \mid v[C] \\
\textit{instructions} & \iota ::= & \texttt{malloc}\, x, \rho, n; \iota \mid x{=}v[i]; \iota \mid v[i]{:=}v'; \iota \mid \texttt{free}\, v; \iota \mid \\
 & & v(v_1, \ldots, v_n) \mid \texttt{halt} \\
\textit{stores} & S ::= & \{\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n\} \\
\textit{programs} & P ::= & (S, \iota)
\end{array}$$

Fig. 1. Language of Locations: Syntax

`malloc` x, ρ, n allocates an unitialized memory block (filled with `junk`) of size n at a new location ℓ, and binds x to the pointer $\texttt{ptr}(\ell)$. The location variable ρ, bound by this instruction, is the static representation of the dynamic location ℓ. The instruction $x{=}v[i]$ binds x to the ith component of the memory block pointed to by v in the remaining instructions. The instruction $v[i]{:=}v'$ stores v' in the ith component of the block pointed to by v. The final memory management primitive, `free` v, deallocates the storage pointed to by v. If v is the pointer $\texttt{ptr}(\ell)$ then deallocation is modeled by updating the store (S) so that the location ℓ maps to `junk`.

The program $(\{\}, \texttt{malloc}\, x, \rho, 2; x[1]{:=}3; x[2]{:=}5; \texttt{free}\, x; \texttt{halt}\,)$ allocates, initializes and finally deallocates a pair of integers. Its evaluation is shown below:

Store	Instructions	
$\{\,\}$	$\texttt{malloc}\, x, \rho, n$	(* allocate new location ℓ, *)
		(* substitute $\texttt{ptr}(\ell), \ell$ for x, ρ *)
$\{\ell \mapsto \langle \texttt{junk}, \texttt{junk} \rangle\}$	$\texttt{ptr}(\ell)[1]{:=}3$	(* initialize field 1 *)
$\{\ell \mapsto \langle 3, \texttt{junk} \rangle\}$	$\texttt{ptr}(\ell)[2]{:=}5$	(* initialize field 2 *)
$\{\ell \mapsto \langle 3, 5 \rangle\}$	$\texttt{free}\, \texttt{ptr}(\ell)$	(* free storage *)
$\{\ell \mapsto \texttt{junk}\}$		

A sequence of instructions (ι) ends in either a `halt` instruction, which stops computation immediately, or a function application ($v(v_1, \ldots, v_n)$). In order to simplify the language and its typing constructs, our functions never return. However, a higher-level language that contains call and return statements can be compiled into our language of locations by performing a *continuation-passing style* (CPS) transformation [14, 15]. It is possible to define a direct-style language, but doing so would force us to adopt an awkward syntax that allows functions to return portions of the store. In a CPS style, all control-flow transfers are handled symmetrically by calling a continuation.

Functions are defined using the form $\texttt{fix}\, f[\Delta; C; \Gamma].\iota$. These functions are recursive (f may appear in ι). The context $(\Delta; C; \Gamma)$ specifies a pre-condition that must be satisfied before the function can be invoked. The type context Δ binds the set of type variables that can occur free in the term; C is a collection of aliasing constraints that statically approximates a portion of the store; and Γ assigns types to free variables in ι.

To call a polymorphic function, code must first instantiate the type variables in Δ using the value form: $v[\eta]$ or $v[C]$. These forms are treated as values because type application has no computational effect (types and constraints are only used for compile-time checking; they can be erased before executing a program).

$$\begin{array}{ll}
(S, \texttt{malloc}\, x, \rho, n; \iota) & \longmapsto (S\{\ell \mapsto \langle \texttt{junk}_1, \ldots, \texttt{junk}_n \rangle\}, \iota[\ell/\rho][\texttt{ptr}(\ell)/x]) \\
\quad \text{where } \ell \notin S & \\
(S\{\ell \mapsto v\}, \texttt{free}\, \texttt{ptr}(\ell); \iota) & \longmapsto (S\{\ell \mapsto \texttt{junk}\}, \iota) \\
\quad \text{if } v = \langle v_1, \ldots, v_n \rangle & \\
(S\{\ell \mapsto v\}, \texttt{ptr}(\ell)[i]{:=}v'; \iota) & \longmapsto (S\{\ell \mapsto \langle v_1, \ldots, v_{i-1}, v', v_{i+1}, \ldots, v_n \rangle\}, \iota) \\
\quad \text{if } v = \langle v_1, \ldots, v_n \rangle \text{ and } 1 \leq i \leq n & \\
(S\{\ell \mapsto v\}, x{=}\texttt{ptr}(\ell)[i]; \iota) & \longmapsto (S\{\ell \mapsto v\}, \iota[v_i/x]) \\
\quad \text{if } v = \langle v_1, \ldots, v_n \rangle \text{ and } 1 \leq i \leq n & \\
(S, v(v_1, \ldots, v_n)) & \longmapsto (S, \iota[c_1, \ldots, c_m/\beta_1, \ldots, \beta_m][v', v_1, \ldots, v_n/f, x_1, \ldots, x_n]) \\
\quad \text{if } v = v'[c_1, \ldots, c_m] & \\
\quad \text{and } v' = \texttt{fix}\, f[\Delta; C; x_1{:}\tau_1, \ldots, x_n{:}\tau_n].\iota & \\
\quad \text{and } \mathit{Dom}(\Delta) = \beta_1, \ldots, \beta_m \qquad \text{(where } \beta \text{ ranges over } \rho \text{ and } \epsilon\text{)} &
\end{array}$$

Fig. 2. Language of Locations: Operational Semantics

3.2 Type Constructors

There are three kinds of type constructors: locations[3] (η), types (τ), and aliasing constraints (C). The simplest types are the base types, which we have chosen to be integers (int). A pointer to a location η is given the singleton type $\mathit{ptr}(\eta)$. The only value in the type $\mathit{ptr}(\eta)$ is the pointer $\texttt{ptr}(\eta)$, so if v_1 and v_2 both have type $\mathit{ptr}(\eta)$, then they must be aliases. Memory blocks have types ($\langle \tau_1, \ldots, \tau_n \rangle$) that describe their contents.

A collection of constraints, C, establishes the connection between pointers of type $\mathit{ptr}(\eta)$ and the contents of the memory blocks they point to. The main form of constraint, written $\{\eta \mapsto \tau\}$, models a store with a single location η containing a value of type τ. Collections of constraints are constructed from more primitive constraints using the join operator ($\oplus$). The empty constraint is denoted by $\emptyset$. We often abbreviate $\{\eta \mapsto \tau\} \oplus \{\eta' \mapsto \tau'\}$ with $\{\eta \mapsto \tau, \eta' \mapsto \tau'\}$.

[3] We use the meta-variable ℓ to denote concrete locations, ρ to denote location *variables*, and η to denote either.

3.3 Static Semantics

Store Typing The central invariant maintained by the type system is that the current constraints C are a faithful description of the current store S. We write this *store-typing invariant* as the judgement $\vdash S : C$. Intuitively, whenever a location ℓ contains a value v of type τ, the constraints should specify that location ℓ maps to τ (or an equivalent type τ'). Formally:

$$\frac{\cdot;\cdot \vdash_v v_1 : \tau_1 \qquad \cdots \qquad \cdot;\cdot \vdash_v v_n : \tau_n}{\vdash \{\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n\} : \{\ell_1 \mapsto \tau_1, \ldots, \ell_n \mapsto \tau_n\}}$$

where for $1 \leq i \leq n$, the locations ℓ_i are all distinct. And,

$$\frac{\vdash S : C' \qquad \cdot \vdash C' = C}{\vdash S : C}$$

Instruction Typing Instructions are type checked in a context $\Delta; C; \Gamma$. The judgement $\Delta; C; \Gamma \vdash_\iota \iota$ states that the instruction sequence is well-formed. A related judgement, $\Delta; \Gamma \vdash_v v : \tau$, ensures that the value v is well-formed and has type τ. [4]

Our presentation of the typing rules for instructions focuses on how each rule maintains the store-typing invariant. With this invariant in mind, consider the rule for projection:

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash_v v : \mathit{ptr}(\eta) \\ \Delta \vdash C = C' \oplus \{\eta \mapsto \langle\tau_1, \ldots, \tau_n\rangle\} \qquad \Delta; C; \Gamma, x{:}\tau_i \vdash_\iota \iota\end{array}}{\Delta; C; \Gamma \vdash_\iota x{=}v[i]; \iota} \begin{pmatrix} x \notin \Gamma \\ 1 \leq i \leq n \end{pmatrix}$$

The first pre-condition ensures that v is a pointer. The second uses C to determine the contents of the location pointed to by v. More precisely, it requires that C equal a store description $C' \oplus \{\eta \mapsto \langle\tau_1, \ldots, \tau_n\rangle\}$. (Constraint equality uses Δ to denote the free type variables that may appear on the right-hand side.) The store is unchanged by the operation so the final pre-condition requires that the rest of the instructions be well-formed under the same constraints C.

Next, examine the rule for the assignment operation:

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash_v v : \mathit{ptr}(\eta) \qquad \Delta; \Gamma \vdash_v v' : \tau' \\ \Delta \vdash C = C' \oplus \{\eta \mapsto \langle\tau_1, \ldots, \tau_n\rangle\} \qquad \Delta; C' \oplus \{\eta \mapsto \tau_{\mathrm{after}}\}; \Gamma \vdash_\iota \iota\end{array}}{\Delta; C; \Gamma \vdash_\iota v[i]{:=}v'; \iota} \ (1 \leq i \leq n)$$

where τ_{after} is $\langle\tau_1, \ldots, \tau_{i-1}, \tau', \tau_{i+1}, \ldots, \tau_n\rangle$

Once again, the value v must be a pointer to some location η. The type of the contents of η are given in C and must be a block with type $\langle\tau_1, \ldots, \tau_n\rangle$. This time the store has changed, and the remaining instructions are checked under the appropriately modified constraint $C' \oplus \{\eta \mapsto \tau_{\mathrm{after}}\}$.

[4] The subscripts on $\vdash_v$ and $\vdash_\iota$ are used to distinguish judgement forms and for no other purpose.

How can the type system ensure that the new constraints $C' \oplus \{\eta \mapsto \tau_{\text{after}}\}$ correctly describe the store? If v' has type τ' and the contents of the location η originally has type $\langle \tau_1, \ldots, \tau_n \rangle$, then $\{\eta \mapsto \tau_{\text{after}}\}$ describes the contents of the location η after the update accurately. However, we must avoid a situation in which C' continues to hold an outdated type for the contents of the location η. This task may appear trivial: Search C' for all occurrences of a constraint $\{\eta \mapsto \tau\}$ and update all of the mappings appropriately. Unfortunately, in the presence of location polymorphism, this approach will fail. Suppose a value is stored in location ρ_1 and the current constraints are $\{\rho_1 \mapsto \tau, \rho_2 \mapsto \tau\}$. We cannot determine whether or not ρ_1 and ρ_2 are aliases and therefore whether the final constraint set should be $\{\rho_1 \mapsto \tau', \rho_2 \mapsto \tau'\}$ or $\{\rho_1 \mapsto \tau', \rho_2 \mapsto \tau\}$.

Our solution uses a technique from the literature on linear type systems. Linear type systems prevent duplication of assumptions by disallowing uses of the contraction rule. We use an analogous restriction in the definition of constraint equality: The join operator $\oplus$ is associative, and commutative, but *not* idempotent. By ensuring that linear constraints cannot be duplicated, we can prove that ρ_1 and ρ_2 from the example above cannot be aliases. The other equality rules are unsurprising. The empty constraint collection is the identity for $\oplus$ and equality on types τ is syntactic up to α-conversion of bound variables and modulo equality on constraints. Therefore:

$$\Delta \vdash \{\rho_1 \mapsto \langle \mathit{int} \rangle\} \oplus \{\rho_2 \mapsto \langle \mathit{bool} \rangle\} = \{\rho_2 \mapsto \langle \mathit{bool} \rangle\} \oplus \{\rho_1 \mapsto \langle \mathit{int} \rangle\}$$

but,

$$\Delta \not\vdash \{\rho_1 \mapsto \langle \mathit{int} \rangle\} \oplus \{\rho_2 \mapsto \langle \mathit{bool} \rangle\} = \{\rho_1 \mapsto \langle \mathit{int} \rangle\} \oplus \{\rho_1 \mapsto \langle \mathit{int} \rangle\} \oplus \{\rho_2 \mapsto \langle \mathit{bool} \rangle\}$$

Given these equality rules, we can prove that after an update of the store with a value with a new type, the store typing invariant is preserved:

Lemma 1 (Store Update). *If* $\vdash S\{\ell \mapsto v\} : C \oplus \{\ell \mapsto \tau\}$ *and* $\cdot;\cdot \vdash_v v' : \tau'$ *then* $\vdash S\{\ell \mapsto v'\} : C \oplus \{\ell \mapsto \tau'\}$.

where $S\{\ell \mapsto v\}$ denotes the store S extended with the mapping $\ell \mapsto v$ (provided ℓ does not already appear on the left-hand side of any elements in S).

Function Typing The rule for function application $v(v_1, \ldots, v_n)$ is the rule one would expect. In general, v will be a value of the form $v'[c_1] \cdots [c_n]$ where v' is a function polymorphic in locations and constraints and the type constructors c_1 through c_n instantiate its polymorphic variables. After substituting c_1 through c_n for the polymorphic variables, the current constraints must equal the constraints expected by the function v. This check guarantees that the no-duplication property is preserved across function calls. To see why, consider the polymorphic function *foo* where the type context Δ is $(\rho_1, \rho_2, \epsilon)$ and the constraints C are $\epsilon \oplus \{\rho_1 \mapsto \langle \mathit{int} \rangle, \rho_2 \mapsto \langle \mathit{int} \rangle\}$:

```
fix foo[Δ; C; x:ptr(ρ1), y:ptr(ρ2), cont:∀[·;ε].(int)→0].
      free x;     (* constraints = ε ⊕ {ρ2 ↦ ⟨int⟩}          *)
      z=y[0];     (* ok because y : ptr(ρ2) and {ρ2 ↦ ⟨int⟩} *)
      free y;     (* constraints = ε                         *)
      cont(z)     (* return/continue                         *)
```

This function deallocates its two arguments, x and y, before calling its continuation with the contents of y. It is easy to check that this function type-checks, but should it? If *foo* is called in a state where ρ_1 and ρ_2 are aliases, a run-time error will result when the second instruction is executed because the location pointed to by y will already have been deallocated. Fortunately, our type system guarantees that *foo* can never be called from such a state.

Suppose that the store currently contains a single integer reference: $\{\ell \mapsto \langle 3 \rangle\}$. This store can be described by the constraints $\{\ell \mapsto \langle \mathit{int} \rangle\}$. If the programmer attempts to instantiate both ρ_1 and ρ_2 with the same label ℓ, the function call $\mathit{foo}[\ell, \ell, \emptyset](\mathtt{ptr}(\ell))$ will fail to type check because the constraints $\{\ell \mapsto \langle \mathit{int} \rangle\}$ do not equal the pre-condition $\emptyset \oplus \{\ell \mapsto \langle \mathit{int} \rangle, \ell \mapsto \langle \mathit{int} \rangle\}$.

Figure 3 contains the typing rules for values and instructions. Note that the judgement $\Delta \vdash_{wf} \tau$ indicates that Δ contains the free type variables in τ.

3.4 Soundness

Our typing rules enforce the property that well-typed programs cannot enter *stuck states.* A state (S, ι) is stuck when no reductions of the operational semantics apply and $\iota \neq \mathtt{halt}$. The following theorem captures this idea formally:

Theorem 1 (Soundness) *If* $\vdash S : C$ *and* $\cdot; C; \cdot \vdash_\iota \iota$ *and* $(S, \iota) \longmapsto \ldots \longmapsto (S', \iota')$ *then* (S', ι') *is not a stuck state.*

We prove soundness syntactically in the style of Wright and Felleisen [28]. The proof appears in the companion technical report [19].

4 Non-linear Constraints

Most linear type systems contain a class of non-linear values that can be used in a completely unrestricted fashion. Our system is similar in that it admits non-linear constraints, written $\{\eta \mapsto \tau\}^\omega$. They are characterized by the axiom:

$$\Delta \vdash \{\eta \mapsto \tau\}^\omega = \{\eta \mapsto \tau\}^\omega \oplus \{\eta \mapsto \tau\}^\omega$$

Unlike the constraints of the previous section, non-linear constraints may be duplicated. Therefore, it is not sound to deallocate memory described by non-linear constraints or to use it at different types. Because there are strictly fewer operations on non-linear constraints than linear constraints, there is a natural subtyping relation between the two: $\{\eta \mapsto \tau\} \leq \{\eta \mapsto \tau\}^\omega$. We extend the subtyping relationship on single constraints to collections of constraints with rules for reflexivity, transitivity, and congruence. For example, assume *add* has type $\forall[\rho_1, \rho_2, \epsilon; \{\rho_1 \mapsto \langle \mathit{int} \rangle\}^\omega \oplus \{\rho_2 \mapsto \langle \mathit{int} \rangle\}^\omega \oplus \epsilon].(\mathit{ptr}(\rho_1), \mathit{ptr}(\rho_2)) \rightarrow \mathtt{0}$ and consider this code:

Instructions	Constraints (Initially $\emptyset$)
$\mathtt{malloc}\ x, \rho, 1;$	$C_1 = \{\rho \mapsto \langle \mathit{junk} \rangle\}$, $x : \mathit{ptr}(\rho)$
$x[0]{:=}3;$	$C_2 = \{\rho \mapsto \langle \mathit{int} \rangle\}$
$\mathit{add}[\rho, \rho, \emptyset](x, x)$	$C_2 \leq \{\rho \mapsto \langle \mathit{int} \rangle\}^\omega = \{\rho \mapsto \langle \mathit{int} \rangle\}^\omega \oplus \{\rho \mapsto \langle \mathit{int} \rangle\}^\omega \oplus \emptyset$

Typing rules for non-linear constraints are presented in Figure 4.

$$\boxed{\Delta;\Gamma\vdash_v v:\tau}$$

$$\frac{}{\Delta;\Gamma\vdash_v i:\mathit{int}} \qquad \frac{}{\Delta;\Gamma\vdash_v x:\Gamma(x)} \qquad \frac{}{\Delta;\Gamma\vdash_v \mathtt{junk}:\mathit{junk}}$$

$$\frac{\Delta\vdash_{wf}\eta}{\Delta;\Gamma\vdash_v \mathtt{ptr}(\eta):\mathit{ptr}(\eta)} \qquad \frac{\Delta;\Gamma\vdash_v v_1:\tau_1 \qquad \cdots \qquad \Delta;\Gamma\vdash_v v_n:\tau_n}{\Delta;\Gamma\vdash_v \langle v_1,\ldots,v_n\rangle:\langle\tau_1,\ldots,\tau_n\rangle}$$

$$\frac{\begin{array}{c}\Delta\vdash_{wf}\forall[\Delta';C].(\tau_1,\ldots,\tau_n)\rightarrow\mathsf{0}\\ \Delta,\Delta';C;\Gamma,f{:}\forall[\Delta';C].(\tau_1,\ldots,\tau_n)\rightarrow\mathsf{0},x_1{:}\tau_1,\ldots,x_n{:}\tau_n\vdash_\iota\iota\end{array}}{\Delta;\Gamma\vdash_v \mathtt{fix}\,f[\Delta';C;x_1{:}\tau_1,\ldots,x_n{:}\tau_n].\iota:\forall[\Delta';C].(\tau_1,\ldots,\tau_n)\rightarrow\mathsf{0}}\ (f,x_1,\ldots,x_n\notin\Gamma)$$

$$\frac{\Delta\vdash_{wf}\eta \qquad \Delta;\Gamma\vdash_v v:\forall[\rho,\Delta';C].(\tau_1,\ldots,\tau_n)\rightarrow\mathsf{0}}{\Delta;\Gamma\vdash_v v[\eta]:\forall[\Delta';C].(\tau_1,\ldots,\tau_n)\rightarrow\mathsf{0}[\eta/\rho]}$$

$$\frac{\Delta\vdash_{wf}C \qquad \Delta;\Gamma\vdash_v v:\forall[\epsilon,\Delta;C'].(\tau_1,\ldots,\tau_n)\rightarrow\mathsf{0}}{\Delta;\Gamma\vdash_v v[C]:\forall[\Delta;C'].(\tau_1,\ldots,\tau_n)\rightarrow\mathsf{0}[C/\epsilon]} \qquad \frac{\Delta;\Gamma\vdash_v v:\tau' \qquad \Delta\vdash\tau'=\tau}{\Delta;\Gamma\vdash_v v:\tau}$$

$$\boxed{\Delta;C;\Gamma\vdash_\iota\iota}$$

$$\frac{\Delta,\rho;C\oplus\{\rho\mapsto\langle\mathit{junk}_1,\ldots,\mathit{junk}_n\rangle\};\Gamma,x{:}\mathit{ptr}(\rho)\vdash_\iota\iota}{\Delta;C;\Gamma\vdash_\iota \mathtt{malloc}\ x,\rho,n;\iota}\ (x\notin\Gamma,\rho\notin\Delta)$$

$$\frac{\begin{array}{c}\Delta;\Gamma\vdash_v v:\mathit{ptr}(\eta)\\ \Delta\vdash C=C'\oplus\{\eta\mapsto\langle\tau_1,\ldots,\tau_n\rangle\} \qquad \Delta;C'\oplus\{\eta\mapsto\mathit{junk}\};\Gamma\vdash_\iota\iota\end{array}}{\Delta;C;\Gamma\vdash_\iota \mathtt{free}\ v;\iota}$$

$$\frac{\begin{array}{c}\Delta;\Gamma\vdash_v v:\mathit{ptr}(\eta) \qquad \Delta\vdash C=C'\oplus\{\eta\mapsto\langle\tau_1,\ldots,\tau_n\rangle\}\\ \Delta;\Gamma\vdash_v v':\tau' \quad \Delta;C'\oplus\{\eta\mapsto\langle\tau_1,\ldots,\tau_{i-1},\tau',\tau_{i+1},\ldots,\tau_n\rangle\};\Gamma\vdash_\iota\iota\end{array}}{\Delta;C;\Gamma\vdash_\iota v[i]{:=}v';\iota}\ (1\leq i\leq n)$$

$$\frac{\begin{array}{c}\Delta;\Gamma\vdash_v v:\mathit{ptr}(\eta')\\ \Delta\vdash C=C'\oplus\{\eta'\mapsto\langle\tau_1,\ldots,\tau_n\rangle\} \qquad \Delta;C;\Gamma,x{:}\tau_i\vdash_\iota\iota\end{array}}{\Delta;C;\Gamma\vdash_\iota x{=}v[i];\iota}\ \begin{pmatrix}x\notin\Gamma\\ 1\leq i\leq n\end{pmatrix}$$

$$\frac{\begin{array}{c}\Delta;\Gamma\vdash_v v:\forall[\cdot;C'].(\tau_1,\ldots,\tau_n)\rightarrow\mathsf{0} \qquad \Delta\vdash C=C'\\ \Delta;\Gamma\vdash_v v_1:\tau_1 \qquad \cdots \qquad \Delta;\Gamma\vdash_v v_n:\tau_n\end{array}}{\Delta;C;\Gamma\vdash_\iota v(v_1,\ldots,v_n)} \qquad \frac{}{\Delta;C;\Gamma\vdash_\iota \mathtt{halt}}$$

Fig. 3. Language of Locations: Value and Instruction Typing

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash_v v : \mathit{ptr}(\eta) \\ \Delta \vdash C = C' \oplus \{\eta \mapsto \langle \tau_1, \ldots, \tau_n \rangle\}^\omega \qquad \Delta; C; \Gamma, x{:}\tau_i \vdash_\iota \iota\end{array}}{\Delta; C; \Gamma \vdash_\iota x{=}v[i]; \iota} \begin{pmatrix} x \notin \Gamma \\ 1 \leq i \leq n \end{pmatrix}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash_v v : \mathit{ptr}(\eta) \qquad \Delta; \Gamma \vdash_v v' : \tau' \\ \Delta \vdash C = C' \oplus \{\eta \mapsto \langle \tau_1, \ldots, \tau_n \rangle\}^\omega \qquad \Delta \vdash \tau' = \tau_i \qquad \Delta; C; \Gamma \vdash_\iota \iota\end{array}}{\Delta; C; \Gamma \vdash_\iota v[i]{:=}v'; \iota} \; (1 \leq i \leq n)$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash_v v : \forall[\cdot; C'].(\tau_1, \ldots, \tau_n) \to 0 \qquad \Delta \vdash C \leq C' \\ \Delta; \Gamma \vdash_v v_1 : \tau_1 \qquad \cdots \qquad \Delta; \Gamma \vdash_v v_n : \tau_n\end{array}}{\Delta; C; \Gamma \vdash_\iota v(v_1, \ldots, v_n)} \qquad \frac{\vdash S : C' \qquad \vdash C' \leq C}{\vdash S : C}$$

Fig. 4. Language of Locations: Non-linear Constraints

4.1 Non-linear Constraints and Dynamic Type Tests

Although data structures described by non-linear constraints cannot be deallocated or used to store objects of varying types, we can still take advantage of the sharing implied by singleton pointer types. More specifically, code can use weak constraints to perform a dynamic type test on a particular object and simultaneously refine the types of many aliases of that object.

To demonstrate this application, we extend the language discussed in the previous section with a simple form of option type $?\langle \tau_1, \ldots, \tau_n \rangle$ (see Figure 5). Options may be `null` or a memory block $\langle \tau_1, \ldots, \tau_n \rangle$. The `mknull` operation associates the name ρ with `null` and the `tosum` v, τ instruction injects the value v (a location containing null or a memory block) into a location for the option type $?\langle \tau_1, \ldots, \tau_n \rangle$. In the typing rules for `tosum` and `ifnull`, the annotation ϕ may either be ω, which indicates a non-linear constraint or $\cdot$, the empty annotation, which indicates a linear constraint.

The `ifnull` v `then` ι_1 `else` ι_2 construct tests an option to determine whether it is `null` or not. Assuming v has type $\mathit{ptr}(\eta)$, we check the first branch (ι_1) with the constraint $\{\eta \mapsto \mathit{null}\}^\phi$ and the second branch with the constraint $\{\eta \mapsto \langle \tau_1, \ldots, \tau_n \rangle\}^\phi$ where $\langle \tau_1, \ldots, \tau_n \rangle$ is the appropriate non-null variant. As before, imagine that *sp* is the stack pointer, which contains an integer option.

```
(* constraints = {η ↦ ⟨ptr(η′)⟩, η′ ↦ ?⟨int⟩}, sp:ptr(η)          *)
r₁=sp[1];             (* r₁:ptr(η′)                                  *)
ifnull r₁ then halt   (* null check                                  *)
else ···              (* constraints = {η ↦ ⟨ptr(η′)⟩}⊕{η′ ↦ ⟨int⟩}^ω *)
```

Notice that a single null test refines the type of multiple aliases; both r_1 and its alias on the stack $sp[1]$ can be used as integer references in the else clause. Future loads of r_1 or its alias will not have to perform a null-check.

These additional features of our language are also proven sound in the companion technical report [19].

Syntax:

$$
\begin{array}{lll}
\textit{types} & \tau ::= \ldots \mid ?\langle \tau_1, \ldots, \tau_n \rangle \mid \textit{null} \\
\textit{values} & v ::= \ldots \mid \mathtt{null} \\
\textit{instructions} & \iota ::= \ldots \mid \mathtt{mknull}\ x, \rho; \iota \mid \mathtt{tosum}\ v, ?\langle \tau_1, \ldots, \tau_n \rangle \mid \\
 & \qquad \mathtt{ifnull}\ v\ \mathtt{then}\ \iota_1\ \mathtt{else}\ \iota_2
\end{array}
$$

Operational semantics:

$$
\begin{array}{ll}
(S, \mathtt{mknull}\ x, \rho; \iota) & \longmapsto (S\{\ell \mapsto \mathtt{null}\}, \iota[\ell/\rho][\mathtt{ptr}(\ell)/x]) \\
\quad \text{where } \ell \notin S & \\
(S, \mathtt{tosum}\ v, ?\langle \tau_1, \ldots, \tau_n \rangle; \iota) & \longmapsto (S, \iota) \\
(S\{\ell \mapsto \mathtt{null}\}, & \\
\quad \mathtt{ifnull}\ \mathtt{ptr}(\ell)\ \mathtt{then}\ \iota_1\ \mathtt{else}\ \iota_2) & \longmapsto (S\{\ell \mapsto \mathtt{null}\}, \iota_1) \\
(S\{\ell \mapsto \langle v_1, \ldots, v_n \rangle\}, & \\
\quad \mathtt{ifnull}\ \mathtt{ptr}(\ell)\ \mathtt{then}\ \iota_1\ \mathtt{else}\ \iota_2) & \longmapsto (S\{\ell \mapsto \langle v_1, \ldots, v_n \rangle\}, \iota_2)
\end{array}
$$

Static Semantics:

$$
\frac{}{\Delta; \Gamma \vdash_v \mathtt{null} : \textit{null}}
\qquad
\frac{\Delta, \rho; C \oplus \{\rho \mapsto \textit{null}\}; \Gamma, x{:}\textit{ptr}(\rho) \vdash_\iota \iota}{\Delta; C; \Gamma \vdash_\iota \mathtt{mknull}\ x, \rho; \iota}\ (x \notin \Gamma, \rho \notin \Delta)
$$

$$
\frac{\begin{array}{cc}\Delta; \Gamma \vdash_v v : \textit{ptr}(\eta) & \Delta \vdash C = C' \oplus \{\eta \mapsto \textit{null}\}^\phi \\ \Delta \vdash_{wf} ?\langle \tau_1, \ldots, \tau_n \rangle & \Delta; C' \oplus \{\eta \mapsto ?\langle \tau_1, \ldots, \tau_n \rangle\}^\phi; \Gamma \vdash_\iota \iota\end{array}}{\Delta; C; \Gamma \vdash_\iota \mathtt{tosum}\ v, ?\langle \tau_1, \ldots, \tau_n \rangle; \iota}
$$

$$
\frac{\begin{array}{c}\Delta; \Gamma \vdash_v v : \textit{ptr}(\eta) \\ \Delta \vdash C = C' \oplus \{\eta \mapsto \langle \tau_1, \ldots, \tau_n \rangle\}^\phi \qquad \Delta; C' \oplus \{\eta \mapsto ?\langle \tau_1, \ldots, \tau_n \rangle\}^\phi; \Gamma \vdash_\iota \iota\end{array}}{\Delta; C; \Gamma \vdash_\iota \mathtt{tosum}\ v, ?\langle \tau_1, \ldots, \tau_n \rangle; \iota}
$$

$$
\frac{\begin{array}{cc}\Delta; \Gamma \vdash_v v : \textit{ptr}(\eta) & \Delta \vdash C = C' \oplus \{\eta \mapsto ?\langle \tau_1, \ldots, \tau_n \rangle\}^\phi \\ \Delta; C' \oplus \{\eta \mapsto \textit{null}\}^\phi; \Gamma \vdash_\iota \iota_1 & \Delta; C' \oplus \{\eta \mapsto \langle \tau_1, \ldots, \tau_n \rangle\}^\phi; \Gamma \vdash_\iota \iota_2\end{array}}{\Delta; C; \Gamma \vdash_\iota \mathtt{ifnull}\ v\ \mathtt{then}\ \iota_1\ \mathtt{else}\ \iota_2}
$$

Fig. 5. Language of Locations: Extensions for option types

5 Related and Future Work

Our research extends previous work on linear type systems [26] and syntactic control of interference [16] by allowing both aliasing and safe deallocation. Several authors [26, 3, 9] have explored alternatives to pure linear type systems to allow greater flexibility. Wadler [26], for example, introduced a new let-form `let!` (x) $y = e_1$ `in` e_2 that permits the variable x to be used as a non-linear value in e_1 (*i.e.* it can be used many times, albeit in a restricted fashion) and then later used as a linear value in e_2. We believe we can encode similar behavior by extending our simple subtyping with bounded quantification. For instance, if a function f requires some collection of aliasing constraints ϵ that are bounded above by $\{\rho_1 \mapsto \langle int \rangle\}^\omega \oplus \{\rho_2 \mapsto \langle int \rangle\}^\omega$, then f may be called with a single linear constraint $\{\rho \mapsto \langle int \rangle\}$ (instantiating both ρ_1 and ρ_2 with ρ and ϵ with $\{\rho \mapsto \langle int \rangle\}$). The constraints may now be used non-linearly within the body of f. Provided f expects a continuation with constraints ϵ, its continuation will retain the knowledge that $\{\rho \mapsto \langle int \rangle\}$ is linear and will be able to deallocate the storage associated with ρ when it is called. However, we have not yet implemented this feature.

Because our type system is constructed from standard type-theoretic building blocks, including linear and singleton types, it is relatively straightforward to implement these ideas in a modern type-directed compiler. In some ways, our new mechanisms simplify previous work. Previous versions of TAL [12, 11] possessed two separate mechanisms for initializing data structures. Uninitialized heap-allocated data structures were stamped with the type at which they would be used. On the other hand, stack slots could be overwritten with values of arbitrary types. Our new system allows us to treat memory more uniformly. In fact, our new language can encode stack types similar to those described by Morrisett *et al.* [11] except that activation records are allocated on the heap rather than using a conventional call stack. The companion technical report [19] shows how to compile a simple imperative language in such a way that it allocates and deletes its own stack frames.

This research is also related to other work on type systems for low-level languages. Work on Java bytecode verification [20, 8] also develops type systems that allows locations to hold values of different types. However, the Java bytecode type system is not strong enough to represent aliasing as we do here.

The development of our language was inspired by the Calculus of Capabilities (CC) [4]. CC provides an alternative to the region-based type system developed by Tofte and Talpin [24]. Because safe region deallocation requires that no aliases be used in the future, CC tracks region aliases. In our new language we adapt CC's techniques to track both object aliases and object type information.

Our work also has close connections with research on alias analyses [5, 21, 17]. Much of that work aims to facilitate program optimizations that require aliasing information in order to be *correct*. However, these optimizations do not necessarily make it harder to check the *safety* of the resulting program. Other work [7, 6] attempts to determine when programs written in unsafe languages, such as C, perform potentially unsafe operations. Our goals are closer to the latter application but differ because we are most interested in compiling *safe* languages and producing low-level code that can be proven safe in a single pass over the program. Moreover, our main result is not a new *analysis* technique, but rather a sound system for representing and checking the results of analysis, and, in particular, for representing aliasing in low-level compiler-introduced data structures rather than for representing aliasing in source-level data.

The language of locations is a flexible framework for reasoning about sharing and destructive operations in a type-safe manner. However, our work to date is only a first step in this area and we are investigating a number of extensions. In particular, we are working on integrating recursive types into the type system as they would allow us to capture regular repeating structure in the store. When we have completed this task, we believe our aliasing constraints will provide us with a safe, but rich and reusable, set of memory abstractions.

Acknowledgements

This work arose in the context of implementing the Typed Assembly Language compiler. We are grateful for the many stimulating discussions that we have had on this topic with Karl Crary, Neal Glew, Dan Grossman, Dexter Kozen, Stephanie Weirich, and Steve Zdancewic. Sophia Drossopoulou, Kathleen Fisher, Andrew Myers, and Anne Rogers gave helpful comments on a previous draft of this work.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Martin Wirsing, editor, *Third International Symposium on Programming Language Implementation and Logic Programming*, pages 1–13, New York, August 1991. Springer-Verlag. Volume 528 of *Lecture Notes in Computer Science.*

[3] Erik Barendsen and Sjaak Smetsers. Conventional and uniqueness typing in graph rewrite systems (extended abstract). In *Thirteenth Conference on the Foundations of Software Technology and Theoretical Computer Science*, pages 41–51, Bombay, 1993. In Shyamasundar, ed., Springer-Verlag, LNCS 761.

[4] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Twenty-Sixth ACM Symposium on Principles of Programming Languages*, pages 262–275, San Antonio, January 1999.

[5] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *ACM Conference on Programming Language Design and Implementation*, pages 230–241, Orlando, June 1994.

[6] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Detecting memory errors via static pointer analysis (preliminary experience). In *ACM Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)*, Montreal, June 1998.

[7] David Evans. Static detection of dynamic memory errors. In *ACM Conference on Programming Language Design and Implementation*, Philadelphia, May 1996.

[8] Stephen N. Freund and John C. Mitchell. A formal framework for the Java bytecode language and verifier. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 147–166, Denver, November 1999.

[9] Naoki Kobayashi. Quasi-linear types. In *Twenty-Sixth ACM Symposium on Principles of Programming Languages*, pages 29–42, San Antonio, January 1999.

[10] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1996.

[11] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based Typed Assembly Language. In *Second International Workshop on Types in Compilation*, pages 95–117, Kyoto, March 1998. Published in Xavier Leroy and Atsushi Ohori, editors, *Lecture Notes in Computer Science*, volume 1473, pages 28-52. Springer-Verlag, 1998.

[12] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, 3(21):528–569, May 1999.

[13] George Necula. Proof-carrying code. In *Twenty-Fourth ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, 1997.

[14] G. D. Plotkin. Call-by-name, call-by-value, and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[15] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Conference Record of the 25th National ACM Conference*, pages 717–740, Boston, August 1972.

[16] John C. Reynolds. Syntactic control of interference. In *Fifth ACM Symposium on Principles of Programming Languages*, pages 39–46, Tucson, 1978.

[17] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, January 1996.

[18] Z. Shao. An overview of the FLINT/ML compiler. In *Workshop on Types in Compilation*, Amsterdam, June 1997. ACM. Published as Boston College Computer Science Dept. Technical Report BCCS-97-03.

[19] Frederick Smith, David Walker, and Greg Morrisett. Alias types. Technical Report TR99-1773, Cornell University, October 1999.

[20] Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. In *Twenty-Fifth ACM Symposium on Principles of Programming Languages*, San Diego, January 1998.

[21] B. Steensgaard. Points-to analysis in linear time. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, January 1996.

[22] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *ACM Conference on Programming Language Design and Implementation*, pages 181–192, Philadelphia, May 1996.

[23] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, November 1990.

[24] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[25] David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *ACM International Conference on Functional Programming and Computer Architecture*, San Diego, CA, June 1995.

[26] Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Programming Concepts and Methods*, Sea of Galilee, Israel, April 1990. North Holland. IFIP TC 2 Working Conference.

[27] A. K. Wright. Simple imperative polymorphism. *LISP and Symbolic Computation*, 8(4), December 1995.

[28] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

Polyvariant Flow Analysis with Constrained Types

Scott F. Smith and Tiejun Wang[*]

Department of Computer Science
The Johns Hopkins University
Baltimore, MD 21218, USA
{scott,wtj}@cs.jhu.edu

Abstract. The basic idea behind improving the quality of a monovariant control flow analysis such as 0CFA is the concept of *polyvariant* analyses such as Agesen's Cartesian Product Algorithm (CPA) and Shivers' nCFA. In this paper we develop a novel framework for polyvariant flow analysis based on Aiken-Wimmers constrained type theory. We develop instantiations of our framework to formalize various polyvariant algorithms, including nCFA and CPA. With our CPA formalization, we show the call-graph based termination condition for CPA will not always guarantee termination. We then develop a novel termination condition and prove it indeed leads to a terminating algorithm. Additionally, we show how data polymorphism can be modeled in the framework, by defining a simple extension to CPA that incorporates data polymorphism.

1 Introduction

The basic idea behind improving the precision of a simple control flow analysis such as 0CFA is the concept of *polyvariant* analysis, also known as *flow splitting*. For better analysis precision, the definition of a polymorphic function is re-analyzed multiple times with respect to different application contexts. The original polyvariant generalization of the monovariant 0CFA control flow algorithm is the nCFA algorithm, defined by Shivers [17]. This generalization however has been shown to be not so effective: the values of n needed to obtain more accurate analyses are usually beyond the realm of the easily computable, and even 1CFA can be quite slow to compute [19]. Better notions of polyvariant analysis have been developed. In particular, Agesen's CPA [1, 2] analyzes programs with parametric polymorphism in an efficient and adaptive manner.

In this paper we develop a general framework for polyvariant flow analysis with Aiken-Wimmers constrained types [3]. We represent each function definition with a polymorphic constrained type scheme of form $(\forall\, \bar{t}.\, t \rightarrow \tau \setminus C)$. The subtyping constraint set C bound in the type scheme captures the flow corresponding to the function body. Each re-analysis of the function is realized by a new instantiation of the type scheme.

There have recently been several frameworks developed for polyvariant flow analysis, in terms of union and intersection types [16], abstract interpretation [13], flow graphs [12], and more implementation-centric [10]. Our purpose in designing a new framework is not primarily to give "yet another framework" for polyvariant flow analysis, but to

[*] Partial funding provided by NSF grant CCR-9619843

G. Smolka (Ed.): ESOP/ETAPS 2000, LNCS 1782, pp. 382–396, 2000.

develop a framework particularly useful for the development of new polyvariant analyses, and for improving on implementations of existing analyses. We will give an example of a new analysis developed within the framework, Data-Adaptive CPA, which extends CPA to incorporate data polymorphism. There also are implementation advantages obtained by basing analyses on polymorphic constrained types. Compared to the flow graph based approach used in other implementations of flow analyses [2, 10, 14], our framework has several advantages: using techniques described in [8, 15], constrained types can be simplified on-the-fly and garbage collection of unreachable constraints can be performed as well, leading to more efficient analyses; and, re-analysis of a function in a different polyvariant context is also realized by instantiation of the function's constrained type scheme, and does not require re-analysis of the function body.

This paper presents the first proposal to use constrained type schemes to model polyvariance; there are several other related approaches in the literature. Palsberg and Pavlopoulou [16] develop an elegant framework for polyvariant analyses in a type system with union/intersection types and subtyping. There are also subtype-free type-based realizations of polymorphism which can be adapted to polyvariant flow analysis. Let-polymorphism is the classic form of polymorphism used in type inference for subtype-free languages, and has been adapted to constrained types in [3, 7], as well as directly in the flow analysis setting by Wright and Jagannathan [19]. Another representation of polymorphism found in subtype-free languages is via rank-2 intersection types [11], which has also been applied to polyvariant flow analysis [4]. The Church group has developed type systems of union/intersection types decorated with flow labels to indicate the flow information [18].

The form of polyvariance we use is quite general: we show how CPA, nCFA, and other analyses may be expressed in the framework. A $\forall$ type is given to each function in the program, and for every different call site and each different type of argument value the function is applied to, a new contour (re-analysis of the function via instantiation of the $\forall$ type) is possible. The framework is flexible in how contours are generated: a completely new contour can be assigned for an particular argument type applied to the function, or for that argument type it can share a pre-existing contour. For example, 0CFA is the strategy which uses exactly one contour for every function.

One difficult problem for CPA is the issue of termination: without a termination check, the analysis may loop forever on some programs, producing infinitely many contours. We develop a termination condition which detects a certain kind of self-referential flow in the constraints and prove that by merging some contours in this case, non-termination is prevented and the analysis is implementable. Our termination condition is different from the call-graph based condition commonly used in other algorithms, which we show will not guarantee termination in all cases.

We also aim here to model polyvariant algorithms capable of handling *data polymorphism*: the ability of an imperative variable to hold values of different types at run-time. Data polymorphism arises quite frequently in object-oriented programming, especially with container classes, and it poses special problems for flow analysis. The one precise algorithm for detecting data polymorphism is the iterative flow analysis (IFA) of Plevyak and Chien [14]. We present a simple non-iterative algorithm, Data-Adaptive CPA, based on an approach distinct from that of IFA.

2 A Framework for Polyvariant Flow Analysis

This section presents the framework of polyvariant constrained type inference. In the next section we instantiate the framework for particular analyses.

2.1 The Language

The language we study here is an extension to the language used in Palsberg and Pavlopoulou's union/intersection type framework for flow analysis [16], adding mutable state so we can model data polymorphism. We believe the concepts of current paper should scale with relative ease to languages with records, objects, classes, and other features, as illustrated in [7, 6].

Definition 21 (The language):

$$e = x \mid n \mid \texttt{succ}\, e \mid \texttt{if0}\, e\, e\, e \mid \lambda x.e \mid e\, e \mid \texttt{new} \mid e \;\texttt{:=}\; e \mid \texttt{!}e \mid e \;\texttt{;}\; e$$

This is a standard call-by-value lambda calculus extended with reference cells. Execution of a `new` expression creates a fresh, uninitialized reference cell. We use `new` because it models the memory creation mode of languages like Java and C++, where uninitialized references are routinely created. Recursive definitions may be constructed in this language via the Y-combinator.

2.2 The Types

Our basis is an Aiken-Wimmers-style constraint system [3]; in particular it is most closely derived from the system described in [7], which combines constraints and mutable state.

Definition 22 (Types): The type grammar is as follows.

$$\begin{array}{lll}
\tau & \in \mathbf{Type} & ::= t \mid \tau v \mid \mathbf{read}\; t \mid \mathbf{write}\; \tau \mid t_1 \to t_2 \\
t & \in \mathbf{TypeVar} \supset \mathbf{ImpTypeVar} & \\
u & \in \mathbf{ImpTypeVar} & \\
\overline{t} & \in \mathbf{TypeVarSet} & = \mathbf{P}_{\mathrm{fin}}(\mathbf{TypeVar}) \\
\tau v & \in \mathbf{ValueType} & ::= \mathbf{int} \mid (\forall\, \overline{t}.\, t \to \tau \setminus C) \mid \mathbf{ref}\; u \\
\tau_1 <: \tau_2 & \in \mathbf{Constraint} & \\
C & \in \mathbf{ConstraintSet} & = \mathbf{P}_{\omega}(\mathbf{Constraint})
\end{array}$$

The types for the most part are standard. Function uses (call sites) are given type $t_1 \to t_2$. $\mathbf{ValueType}$ are the types for data values. $\mathbf{ref}\; u$ is the type for a cell whose content has type u. We distinguish imperative type variables $u \in \mathbf{ImpTypeVar}$ for the presentation of data polymorphism. Read and write operations on reference cells are represented with types $\mathbf{read}\; t$ and $\mathbf{write}\; \tau$ respectively. Functions are given polymorphic types $(\forall\, \overline{t}.\, t \to \tau \setminus C)$, where t is the type variable for the formal argument, τ is the return type, C is the set of constraints bound in this type scheme, and $\overline{t}$ is the set of bound type variables. Such types are also referred as $\forall$ types or closure types in the paper.

$$\text{(Var)}\quad \frac{A(x) = t}{A \vdash x : t \setminus \{\}}$$

$$\text{(Int)}\quad \frac{}{A \vdash n : \mathbf{int} \setminus \{\}}$$

$$\text{(Succ)}\quad \frac{A \vdash e : \tau \setminus C}{A \vdash \texttt{succ}\, e : \mathbf{int} \setminus \{\tau <: \mathbf{int}\} \cup C}$$

$$\text{(If0)}\quad \frac{A \vdash e_1 : \tau_1 \setminus C_1,\ e_2 : \tau_2 \setminus C_2,\ A \vdash e_3 : \tau_3 \setminus C_3}{A \vdash \texttt{if0}\, e_1\, e_2\, e_3 : t \setminus \{\tau_1 <: \mathbf{int},\ \tau_2 <: t,\ \tau_3 <: t\} \cup C_1 \cup C_2 \cup C_3}$$

$$\text{(Abs)}\quad \frac{A, \{x : t\} \vdash e : \tau \setminus C}{A \vdash \lambda x.e : (\forall\, \overline{t}.\, t \to \tau \setminus C) \setminus \{\}}$$

$$\text{where } \overline{t} = \mathit{FreeTypeVar}(t \to \tau \setminus C) - \mathit{FreeTypeVar}(A)$$

$$\text{(Appl)}\quad \frac{A \vdash e_1 : \tau_1 \setminus C_1,\ e_2 : \tau_2 \setminus C_2}{A \vdash e_1\, e_2 : t_2 \setminus \{\tau_1 <: t_1 \to t_2,\ \tau_2 <: t_1\} \cup C_1 \cup C_2}$$

$$\text{(New)}\quad \frac{}{A \vdash \texttt{new} : \mathbf{ref}\ u \setminus \{\}}$$

$$\text{(Read)}\quad \frac{A \vdash e : \tau \setminus C}{A \vdash \texttt{!}e : t \setminus \{\tau <: \mathbf{read}\ t\}}$$

$$\text{(Write)}\quad \frac{A \vdash e_1 : \tau_1 \setminus C_1,\ e_2 : \tau_2 \setminus C_2}{A \vdash e_1 \texttt{:=} e_2 : \tau_2 \setminus \{\tau_1 <: \mathbf{write}\ \tau_2\} \cup C_1 \cup C_2}$$

$$\text{(Seq)}\quad \frac{A \vdash e_1 : \tau_1 \setminus C_1,\ e_2 : \tau_2 \setminus C_2}{A \vdash e_1\ ;\ e_2 : \tau_2 \setminus C_1 \cup C_2}$$

Fig. 1. Type inference rules

2.3 The Type Inference Rules

We present the type inference rules in Figure 1. A type environment A is a mapping from program variables to type variables. Given a type environment A, the proof system assigns a type to expression e via the type judgment $A \vdash e : \tau \setminus C$, where τ is the type for e, and C is the set of constraints which models the flow paths in e. We abbreviate $A \vdash e : \tau \setminus C$ as $\vdash e : \tau \setminus C$ when A is empty. The rules are deterministic except that nondeterminism may arise in the choice of type variables. We restrict type derivations to be of a form where fresh type variables are used whenever it is possible. With this restriction, type inference is trivially decidable and is unique modulo choice of type variable names.

Definition 23 (Type inference algorithm): For closed expression e, its inferred type is $\tau \setminus C$ provided $\vdash e : \tau \setminus C$.

The intuition behind those inference rules is that a subtyping constraint $\tau_1 <: \tau_2$ indicates a potential flow from expressions of type τ_1 to expressions of type τ_2. The rules generally follow standard presentations of Aiken-Wimmer constrained type system, except for the (Abs) and cell typing rules. Detailed descriptions of other rules could be found in [7, 3]. The (Abs) rule assigns each function a polymorphic type $(\forall\ \overline{t}.\ t \to \tau \setminus C)$. In this rule, $\mathit{FreeTypeVar}(\cdot)$ is a function that extracts free type variables, $\overline{t}$ collects all the type variables generated when the inference is applied to the function body, and C collects all the constraints corresponding to the function body. The manner in which $\forall$ type schemes are formed is similar to standard polymorphic constrained type systems, but the significant difference here is that every function is given a $\forall$ type. By contrast, in a system based on let-polymorphism, the `let` construct dictates where $\forall$ types are introduced.

The (New) rule assigns the reference cell type $\mathbf{ref}\ u$, with u, the type of the cell content, initially unconstrained. In the (Read) rule, $\mathbf{read}\ t$ is the type for a cell whose read result is of type t. In the (Write) rule, $\mathbf{write}\ \tau_2$ is the type for a cell assigned with a value of type τ_2.

We take an intensional view of types: two types are equivalent if and only if they are syntactically identical. In particular, $\forall$ types corresponding to different functions in the program are always different, even though they might be α-variants. This is because we wish to distinguish different functions in the analysis to obtain precise flow properties. For type soundness properties, an extensional view could be taken.

We illustrate the inference rules with the example studied in [16]:

$$E_1 \equiv (\lambda f.\mathtt{succ}\,((f\ f)\ 0)\ (\mathtt{if0}\ n\ (\lambda x.x)\ (\lambda y.\lambda z.z))$$

To ease presentation, each program variable is inferred with a type variable having exactly the same name. We have

$\vdash\ (\lambda f.\mathtt{succ}\,((f\ f)\ 0)\ :\ \tau_f\backslash\{\}$, where $\tau_f \equiv (\forall\ \{f, t_1, t_2, t_3, t_4\}.f \rightarrow \mathbf{int} \setminus \{f <: t_1 \rightarrow t_2, f <: t_1, t_2 <: t_3 \rightarrow t_4, \mathbf{int} <: t_3, t_4 <: \mathbf{int}\})$,
$\vdash\ (\lambda x.x) : \tau_x\backslash\{\}$, where $\tau_x \equiv (\forall\ \{x\}.x \rightarrow x\backslash\{\})$,
$\vdash\ (\lambda y.\lambda z.z)\ :\ \tau_y\backslash\{\}$, where $\tau_y \equiv (\forall\ \{y\}.y \rightarrow (\forall\ \{z\}.z \rightarrow z\backslash\{\})\backslash\{\})$.
$\vdash\ E_1\ :\ t_7 \setminus \{\mathbf{int} <: \mathbf{int}, \tau_x <: t_5, \tau_y <: t_5, \tau_f <: t_6 \rightarrow t_7, t_5 <: t_6\}$

2.4 Computation of the Closure

The inference algorithm applied to program e results in a type judgment $\vdash\ e\ :\ \tau \setminus C$. For a flow analysis, we need to generate all the possible data-flow and control-flow paths and propagate value types along all the data-flow paths. This is achieved by applying the closure rules of Figure 2 to C, propagating information via deduction rules on the subtyping constraints.

The rule (Trans) is the transitivity rule which models run-time data flow by propagating value types forward along flow paths. The (Read) closure rule applies when a read operation is applied on a cell of type $\mathbf{ref}\ u$, and the reading result is of type t. By constraint $u <: t$, the cell content flows to the reading result. The (Write) closure rule applies when a write operation is applied on a cell of type $\mathbf{ref}\ u$, and a value of type τ is assigned to the cell. By constraint $\tau <: t$, the value flows to the content of the cell. With (Read) and (Write) rules together, any value assigned to a cell flows to the cell's reading result. Flanagan [9] uses a related set of rules for references and was the source of the idea for us.

The most important closure rule is (Fun), which performs $\forall$ elimination. The constraint $(\forall\ \overline{t}.\ t \rightarrow \tau \setminus C) <: t_1 \rightarrow t_2$ indicates a function flowing to a call site, where $(\forall\ \overline{t}.\ t \rightarrow \tau \setminus C)$ is the type for the function and $t_1 \rightarrow t_2$ is the type representing the call site. The constraint $\tau v <: t_1$ means that a value of type τv flows in as the actual

$$\text{(Trans)}\ \frac{\tau v <: t,\ \ t <: \tau}{\tau v <: \tau}$$

$$\text{(Fun)}\ \ \frac{(\forall\,\overline{t}.\,t \to \tau \setminus C) <: t_1 \to t_2,\ \ \tau v <: t_1}{\tau v <: \Theta(t),\ \ \Theta(\tau) <: t_2,\ \ \Theta(C)}$$

$$\text{where } \Theta = \mathtt{Poly}((\forall\,\overline{t}.\,t \to \tau \setminus C) <: t_1 \to t_2, \tau v)$$

$$\text{(Read)}\ \frac{\mathbf{ref}\ u <: \mathbf{read}\ t}{u <: t}$$

$$\text{(Write)}\ \frac{\mathbf{ref}\ u <: \mathbf{write}\ \tau}{\tau <: u}$$

Fig. 2. Constraint Closure rules

argument. At run-time, upon each function application, all local variables of the function are allocated fresh locations on the stack. To model this behaviour in the analysis, a renaming $\Theta \in \mathbf{TypeVar} \overset{\mathrm{p}}{\to} \mathbf{TypeVar}$ is applied to type variables in $\overline{t}$. The partial function Θ is extended to types, constraints, and constraint sets in the usual manner. $\Theta(t)$ for $t \notin \overline{t}$ is defined to return t. We call $\Theta(\tau)$ an *instantiation* of τ. Following the terminology of Shivers' nCFA [17], we call a renaming Θ a *contour*. The $\forall$ is eliminated from $(\forall\ \overline{t}.\ t \to \tau \setminus C)$ by applying Θ to C. The (Fun) rule then generates additional constraints to capture the flow from the actual argument τv to the formal argument $\Theta(t)$, and from the return value $\Theta(\tau)$ to the application result t_2. The (Fun) rule is parameterized by function $\mathtt{Poly} \in \mathbf{Constraint} \times \mathbf{ValueType} \to (\mathbf{TypeVar} \overset{\mathrm{p}}{\to} \mathbf{TypeVar})$, which decides for this particular function, call site, and actual argument type, which contour is to be used (*i.e.*, created or reused). Providing a concrete `Poly` instantiates the framework to give a concrete algorithm. For example, the monovariant analysis 0CFA is defined by letting `Poly` always return the identity renaming. This particular example shows how `Poly` may reuse existing contours. The differing analyses are defined by differing `Poly` which use different strategies for sharing contours. In the next section we show how this works by presenting some particular `Poly`.

Definition 24 (Closure): For a constraint set C, $\mathit{Closure}_{\mathtt{Poly}}(C)$ is the least superset of C closed under the closure rules of Figure 2.

This closure is well-defined since the rules can be seen to induce a monotone function on constraint sets. By this definition, some `Poly` may produce infinite closures since infinitely many contours may be created. Such analyses are still worthy of study even though they are usually unimplementable.

Definition 25 (Flow Analysis): Define $\mathit{Analysis}_{\mathtt{Poly}}(e) = \mathit{Closure}_{\mathtt{Poly}}(C)$, where the inference algorithm infers $\vdash\ e\ :\ \tau \setminus C$.

The output of an analysis is a set of constraints, which is the closure of the constraint set generated by the inference rules. The closure contains complete flow information about the program, various program properties can be deduced from it.

Definition 26 (Type-Checking): A program e is well-typed iff $Analysis_{\texttt{Poly}}(e)$ contains no immediately type-contradictory constraints such as $\mathbf{ref}\ u <: t \rightarrow t'$.

For example, analyzing program $\texttt{succ}\ (\lambda x.x)$ would generate a type-contradictory constraint $(\forall\ \{x\}.x \rightarrow x \backslash \{\}) <: \mathbf{int}$, which indicates an type error. A computation state is *wrong* if computation cannot continue due to a type error. Our type system does not statically check for errors due to reading uninitialized cells.

To illustrate how the results of a conventional control flow analysis can be obtained in our framework, we use the fact that by the structure of the inference rules, every $\forall$ type in the closure corresponds to a unique lambda abstraction in the program.

Definition 27 (Control Flow Analysis): For an expression e in the program, if e is assigned with type τ by the inference rules, the function corresponding to $(\forall\ \overline{t'}.\ t' \rightarrow \tau' \setminus C')$ is considered flowing to e, if either $\tau = (\forall\ \overline{t'}.\ t' \rightarrow \tau' \setminus C')$ or $(\forall\ \overline{t'}.\ t' \rightarrow \tau' \setminus C') <: t \in Analysis_{\texttt{Poly}}(e)$, and either $\tau = t$ or t is an instantiation of τ.

The above definition includes two cases: either e is directly assigned with a $\forall$ type, in this case e is a lambda abstraction which trivially flows to itself; or e is assigned with a type variable by the inference rules, and the type variable or an instantiation of it has a $\forall$ type as lower bound.

A subject reduction property for our type system can be established, with a proof similar to the one in [7]. The subject reduction property implies the type soundness and flow soundness of the framework.

Theorem 28 (Subject Reduction, Type Soundness, Flow Soundness):

1. The type system has a subject reduction property;
2. A well-typed program e cannot go *wrong* during execution;
3. If an expression evaluates to a closure value of a function, the function is considered flowing to the expression by the the control flow analysis.

The soundness of the framework implies that any analysis defined as an instantiation of the framework is also sound.

3 Instantiating the Framework

In this section we present various polyvariant algorithms as instantiations of our framework.

3.1 nCFA Instantiation

In Shivers' nCFA analysis [17], each function application (call) is associated with a call-string of length at most n. The call-string contains the last n or fewer calls on the call-path leading to this application. Applications of the same function share the same contour (i.e., analysis of the function) if they have the same call-string. To present nCFA in our framework, type variables are defined with superscripts that denote the call-string:

$$\begin{array}{lll} \alpha \in \mathbf{Identifier} & & \\ s \in \mathbf{Superscript} & = & \mathbf{Identifier\ List} \\ t \in \mathbf{TypeVar} & ::= & \alpha^s \end{array}$$

We use the following list notation: The empty list is $[\,]$, $[\alpha_1, \ldots, \alpha_m]$ is a list of m elements, $l_1 \ @ \ l_2$ appends lists l_1 and l_2, and $l(1..n)$ is the list consisting of the first $min(n, length(l))$ elements of list l. Each type variable α^s is tagged with a call-string s. All type variables generated by the inference rules have empty lists as superscripts. By the inference rule (Appl), a call site is inferred with a type $\alpha_1^{[\,]} \to \alpha_2^{[\,]}$, we use α_2 to identify this call site, thus a call-string is a list of such identifiers. All bound type variables of a $\forall$ type have empty list superscripts. When the $\forall$ quantifier is eliminated by the (Fun) closure rule, those bound type variables are renamed by changing the superscripts from empty lists to the appropriate call-strings.

Definition 31 (nCFA Algorithm): The nCFA algorithm is defined as the instantiation of the framework with Poly $=$ **CFA**, where

$$\mathbf{CFA}((\forall\, \overline{t}.\, t \to \tau \setminus C) <: t_1 \to \alpha_2^{s_2}, \tau v) = \Theta, \text{ where for each } \alpha^{[\,]} \in \overline{t},$$
$$\Theta(\alpha^{[\,]}) = \alpha^{s'}, \text{where } s' = ([\alpha_2] \ @ \ s_2)(1..n)$$

It can be shown by induction that s' is the call-string for application $(\forall\ \overline{t}.\ t \to \tau \setminus C) <: t_1 \to \alpha_2^{s_2}$. The definition of Θ ensures that applications of the same function share the same contour if and only if they have the same call-string.

Not only is nCFA inefficient, but even for large n it may be imprecise. Applying nCFA to program E_1, since $(\lambda f \ldots)$ has only one application, the (Fun) rule generates only one contour Θ for this function, resulting in $\tau_x <: \Theta(f)$ and $\tau_y <: \Theta(f)$. This means both $(\lambda x.x)$ and $(\lambda y.\lambda z.z)$ flow to f, and at the application site $f\ f$ there are four applications. One of them, $(\lambda x.x)$ applying to $(\lambda y.\lambda z.z)$ leads to a type error: $(\forall\ \{z\}.z \to z\backslash\{\}) <: \mathbf{int}$. Hence nCFA fails to type-check E_1 for arbitrary n.

3.2 Idealized CPA

The Cartesian Product Algorithm (CPA) [1, 2] is a concrete type inference algorithm for object-oriented languages. For a message sending expression, CPA computes the cartesian product of the types for the actual arguments. For each element of the cartesian product, the method body is analyzed exactly once with one contour generated. The calling-contexts of a method are partitioned by the cartesian product, rather than by call-strings as in nCFA. In our language, each function has only one argument. For each function, CPA generates exactly one contour for each distinct argument type that the function may be applied to. Without a termination check, CPA may fail to terminate for some programs. We first present an idealized CPA which may produce an infinite closure, and in Section 5 show how a terminating CPA analysis may be defined which keeps the closure finite. To present CPA, type variables are defined with structure:

$$\begin{aligned} &\alpha \in \mathbf{Identifier} \\ &t\ \in \mathbf{TypeVar}\ ::= \alpha \mid \alpha^{\tau v} \end{aligned}$$

The inference rules are constrained to generate type variables without superscripts.

Definition 32 (Idealized CPA algorithm): The Idealized CPA algorithm is the instantiation of the framework with Poly $=$ **CPA**, where

$$\mathbf{CPA}((\forall\, \overline{t}.\, t \to \tau \setminus C) <: t_1 \to t_2, \tau v) = \Theta, \text{ where for each } \alpha \in \overline{t}, \Theta(\alpha) = \alpha^{\tau v}$$

The contours Θ are generated based on the actual argument type τv, independent of the application site $t_1 \to t_2$. This is the opposite of **CFA**, which ignores the value type τv, and only uses the call site $t_1 \to t_2$. Given a particular function and its associated $\forall$ type in a program, this algorithm will generate a unique contour ($\forall$ elimination) for each distinct value type the function is applied to. It however may share contours across call sites. Agesen [2] presents convincing experimental evidence that the CPA approach is both more efficient and more feasible than nCFA.

We now sketch what Idealized CPA will produce when applied to program E_1. Even though there is only one application site for $(\lambda f \dots)$, it applies to two different actual argument values. So, the (Fun) rule generates two contours Θ_1 and Θ_2 for $(\lambda f \dots)$ with $\Theta_1(f) = f^{\tau_x}, \tau_x <: f^{\tau_x}, \Theta_2(f) = f^{\tau_y}, \tau_y <: f^{\tau_y}$. At application site $f\, f$, there would be only two applications: $(\lambda x.x)$ applying to itself and $(\lambda y.\lambda z.z)$ applying to itself. Thus the program is type-checked successfully.

4 Data Polymorphism

Data polymorphism is defined informally in [2] as the ability of an imperative program variable to hold values of different types at run-time. In our language, a more precise definition could be that cells created from a single imperative creation point (`new` expression) in the program could be assigned with run-time values of different types. CPA addresses parametric polymorphism effectively, but may lose precision in the presence of data polymorphism. For example, consider when CPA is applied to the program

$$E_2 \equiv (\lambda f.(\lambda x.\ x \mathrel{\texttt{:=}} 0; \texttt{succ}\ \texttt{!}x)(f\,1); (f\,2) \mathrel{\texttt{:=}} (\lambda y.y))\,(\lambda z.\texttt{new})$$

Function $(\lambda y.y)$ has type $(\forall\ \{y\}.y \to y\backslash\{\})$, and $(\lambda z.\texttt{new})$ has type $(\forall\ \{z, u\}.z \to \mathbf{ref}\ u\backslash\{\})$. The two applications of $(\lambda z.\texttt{new})$ have same actual argument type **int**, so one contour Θ is shared by the two applications. At run-time the two applications return two distinct cells, but in CPA closure, the two cells share type $\mathbf{ref}\ u'$ (assume $\Theta(u) = u'$), since there is only one contour for $(\lambda z.\texttt{new})$. At run-time, one cell is assigned with 0, and the other is assigned with $(\lambda y.y)$. The two assignments are both reflected on u' as constraints $\mathbf{int} <: u'$ and $(\forall\ \{y\}.y \to y\backslash\{\}) <: u'$, as if there were only one cell, which is assigned with values of two different types. This leads to a type error: $(\forall\ \{y\}.y \to y\backslash\{\}) <: \mathbf{int}$. But if distinct contours were used for the two applications of $(\lambda z.\texttt{new})$, the two cells would have separate cell types and the program would be type-checked.

This small example illustrates that data polymorphism is a problem that arises in a function that contains a creation point (a `new` expression). Different applications of the function may create different cells which are assigned with values of different types, a precise analysis should disambiguate these cells by letting them have separate cell types. To illustrate how data polymorphism can be modeled in our framework, we present a refinement of CPA to give better precision in the analysis of data polymorphic programs.

Consider two applications of a single function. If the applications have same actual argument type, then CPA generates a single contour for them. But, if the two applications return separate mutable data structures at run-time, and the data structures are modified differently after being returned from the two different applications, CPA would lose precision by merging the data structures together. If two separate contours were used for the two applications, the imprecision could be avoided. In the result of CPA analysis, such a function has a return type which is a mutable data structure with polymorphic contents. We call such functions *data-polymorphic*. In program E_2, $(\lambda z.\texttt{new})$ is data-polymorphic, and the other functions are data-monomorphic.

Based on the above observation, our Data-Adaptive CPA algorithm is a two-pass analysis. The first pass is just CPA. From the CPA closure, we detect a set of functions which are possibly data-polymorphic. In the second pass, for data-polymorphic functions, a distinct contour is generated for *every* function application. In this way, imprecision associated with data-polymorphic functions can be avoided; only CPA splitting is performed for data-monomorphic functions, avoiding generation of redundant contours.

Mutable data structures with polymorphic contents are detected from the CPA closure with following definition:

Definition 41 (Data Polymorphic Types): Type τ is data-polymorphic in constraint set C if any of the following cases hold:

1. $\tau = \mathbf{ref}\ u$, $\tau v_1 <: u \in C$, $\tau v_2 <: u \in C$, and $\tau v_1 \neq \tau v_2$;
2. τ is type variable t, $\tau' <: t \in C$, and τ' is data-polymorphic in C;
3. $\tau = \mathbf{ref}\ u$ and u is data-polymorphic in C;
4. $\tau = (\forall\ \overline{t'}.\ t' \to \tau' \setminus C')$ and τ' is data-polymorphic in C.

The above definition is inductive. The first case is the base case, detecting cell types with polymorphic contents. The second case declares a type variable as data-polymorphic when it has a data polymorphic lower bound. The remaining two cases are inductive cases based on the idea that a type is data-polymorphic if it has a data-polymorphic component. Particularly, a closure type is declared as data-polymorphic when the type of its return value is data-polymorphic. Note that, for purely functional programs with no usage of cells, no types would be detected as data-polymorphic.

Recall that CPA type variables are either of the form α or $\alpha^{\tau v}$. We define an operation $erase$ on type variables as: $erase(\alpha) = \alpha$, $erase(\alpha^{\tau v}) = \alpha$. And we extend it naturally to types, defining $erase(\tau)$ as the type with all superscripts erased from all type variables in τ. In particular, $erase$ maps a closure type to the type for the lambda abstraction in the program corresponding to the closure type, and it maps a cell type to the type for the corresponding creation point (`new` expression) in the program. From now on, we call τv an *instantiation* of $erase(\tau v)$.

Definition 42 (Data Polymorphic Functions): For function $\lambda x.e$ assigned with type $(\forall\ \overline{t}.\ t \to \tau \setminus C)$ by the inference rules, $\lambda x.e$ is a data-polymorphic function in constraint set C' iff there appears τ' in C' s.t. $erase(\tau') = \tau$ and τ' is data-polymorphic in C'.

In the above definition we use the fact that every distinct function in the program is given a unique type $(\forall\ \overline{t}.\ t \to \tau \setminus C)$ by the inference rules. The constraint set C' is a flow analysis result of the program. The condition $erase(\tau') = \tau$ means that the function $\lambda x.e$ may return a value of type τ'. Since τ' is data-polymorphic in C', we know that, according to analysis result C', the function may return mutable data structures with polymorphic contents, and we declare it as a data-polymorphic function.

Definition 43 (Data-Adaptive CPA): For program e, Data-Adaptive CPA is an instantiation of the framework with `Poly` $= \mathbf{DCPA}$, where

$$\mathbf{DCPA}((\forall\,\overline{t}.\,t \to \tau \setminus C) <: t_1 \to t_2, \tau v) = \Theta, \text{where for each } \alpha \in \overline{t},$$
$$\Theta(\alpha) = \begin{cases} \alpha' & \text{where } \alpha' \text{ is a fresh identifier, if } \mathit{erase}(\forall\,\overline{t}.\,t \to \tau \setminus C) \text{ is type for} \\ & \text{a data-polymorphic function in } \mathit{Analysis}_{\mathbf{CPA}}(e) \\ \alpha^{\tau v} & \text{otherwise} \end{cases}$$

The second pass of Data-Adaptive CPA differs from CPA only when the function is detected as data polymorphic in the closure obtained by the first CPA pass. In this case, a new contour is always generated for every application. We now illustrate Data-Adaptive CPA on program E_2. After the first CPA pass, we have

$$\mathbf{int} <: u', (\forall\ \{y\}.y \to y\backslash\{\}) <: u') \in \mathit{Analysis}_{\mathbf{CPA}}(E_2)$$

Thus u' is data-polymorphic, and so is $\mathbf{ref}\ u'$. Since $\lambda z.$`new` is inferred with type $(\forall\ \{z, u\}.z \to \mathbf{ref}\ u\backslash\{\})$ and $\mathit{erase}(\mathbf{ref}\ u') = \mathbf{ref}\ u$, $\lambda z.$`new` is a data-polymorphic function. In the second pass, the two applications of $\lambda z.$`new` have separate contours, and the program type-checks.

We briefly sketch how Data-Adaptive CPA could be applied to data polymorphism in object-oriented programming. We illustrate the ideas by assuming an encoding of instance variables as cells, objects as records (which we expect can be added to our language without great difficulty), classes as class functions, and object creation as application of class functions. An example of such an encoding is presented in [7].

Consider applying such an encoding to the Java program fragment of Figure 3: The

```
class Box {
    public Object content;
    public void set(Object obj) {
        content=obj;
    }
    public Object get() {
        return content;
    }
}
...
Box box1=new Box(); box1.set(new Integer(0));
Box box2=new Box(); box2.set(new Boolean(true));
... box1.get() ...
```

Fig. 3. Java program exhibiting the need for data polymorphism

two `new Box()` expressions would be encoded as two applications of the class function for class `Box`. When CPA is applied, since the two applications always apply to arguments of same type in any object encoding, the two applications share a single contour. Thus the two `Box` instances share a same object type, and the analysis would imprecisely conclude that the result of `box1.get()` includes object type `Boolean`. When Data-Adaptive CPA is applied, from the closure of the first CPA pass, the instance variable `content` would be detected as being associated with a data-polymorphic cell type. Since the class function for class `Box` returns a object value with `content` as a component, the class function would be detected as a data-polymorphic function. During the second pass, the two applications of the class function would have separate contours, thus the two instances of `Box` would have separate types and the imprecision would be avoided.

For programs with much data polymorphism, Data-Adaptive CPA may become impractical as many functions are detected as data-polymorphic. Similar to Agesen's CPA implementation [2], a practical implementation should restrict the number of contours generated.

Plevyak and Chien's iterative flow analysis (IFA) [14] uses an iterative approach for precise analysis of data polymorphic programs. The first pass analyzes the program by letting objects of the same class share the same object contour. Every pass detects a set of confluence points (imprecise flow graph nodes where different data values merge) based on the result of the previous pass, and generates more contours with aim to resolve the imprecision at confluence points. The iteration continues until a fixed-point is reached. The advantage of IFA is that splitting is performed only when it is profitable, yet every pass is a whole-program analysis and the number of passes needed could be large. Use of declarative parametric polymorphism [5] to guide the analysis of data polymorphism could be a completely different approach that also could be considered.

5 Terminating CPA Analyses

Any instantiation of our polyvariant framework terminates when only finitely many distinct contours are generated. The nCFA algorithms we defined terminate for arbitrary programs since the number of call-strings of length no more than n is finite. Unfortunately, the Idealized CPA and Data-Adaptive CPA algorithms fail to terminate for some programs.

Agesen [2] develops various methods to detect recursion and avoid the generation of infinitely many contours over recursive functions in his CPA implementation. One approach is to construct a call-graph during analysis, and restrict the number of contours generated along a cycle in the call-graph. However, for Idealized CPA, adding call-graph cycle detection is not enough to ensure termination. Consider the program

$$E_3 \equiv (\lambda c.\ c \mathrel{\texttt{:=}} \lambda x.x; (\lambda d.\ c \mathrel{\texttt{:=}} (\lambda y.\ d\ y))\ \texttt{!}c)\ \texttt{new}$$

Its call-graph has only one edge: function $(\lambda c \ldots)$ calls $(\lambda d \ldots)$. There is no cycle in it. Consider running Idealized CPA on the program. For each value type lower bound of of u (assume the cell has type $\mathbf{ref}\ u$), there is a contour generated for function $(\lambda d \ldots)$. At first the type for $f_0 = (\lambda x.x)$ becomes a lower bound of u, one contour is generated for function $(\lambda d \ldots)$, and the type for closure $f_1 = (\lambda y.f_0\ y)$ becomes another lower bound of u. So another contour is generated for $(\lambda d \ldots)$, and the type for closure $f_2 = (\lambda y.f_1\ y)$ also becomes a lower bound of u. This process would repeat forever, with an infinite number of contours generated for function $(\lambda d \ldots)$. This example shows that call-graph based approach cannot ensure the termination of Idealized CPA.

Here we present a novel approach that ensures the termination of CPA for arbitrary programs. Our approach is based on the following observation: when Idealized CPA fails to terminate for a program, there must be a cyclic dependency relation among those functions having infinitely many contours. In the example, there exists such a cyclic relation: function $(\lambda y \ldots)$ is lexically enclosed by $(\lambda d \ldots)$, and $(\lambda d \ldots)$ applies to closure values corresponding to $(\lambda y \ldots)$. If we detect such cycles and restrict the number of contours generated for functions appearing in cycles, non-termination could be avoided. To be precise, the key of our method is to construct a relation among value types during closure computation. This relation is defined as:

Definition 51 (Flow Dependency, $\Rightarrow$): For constraint set C, define $\Rightarrow$ as a relation among value types such that if either

$$\tau v_1 <: t \rightarrow t', \tau v_2 <: t \in C, \tau v_1 = (\forall\, \overline{t}_1.\, t_1 \rightarrow \tau_1 \setminus C_1)$$

or

τv_1 occurs as a subterm of $(\forall\, \overline{t}_2.\, t_2 \rightarrow \tau_2 \setminus C_2), \tau v_2 = (\forall\, \overline{t}_2.\, t_2 \rightarrow \tau_2 \setminus C_2), \tau v_1 \neq \tau v_2$, and there exists a $t \in \overline{t}_2$ such that t appears in τv_1

holds, then $erase(\tau v_1) \Rightarrow erase(\tau v_2)$ in C.

The first case above defines a dependency when closure type τv_1 applies to value type τv_2. The second case defines a dependency when closure type τv_2 contains value type τv_1 as a subterm, so that when a new contour is generated for closure type τv_2, a new value type is created which is τv_1 with some of its free type variables renamed. If $\tau v_1 \Rightarrow \tau v_2$ in C_1, and $C_1 \subseteq C_2$, we have $\tau v_1 \Rightarrow \tau v_2$ in C_2. Thus relation $\Rightarrow$ could be incrementally computed along with the incremental closure computation. We abbreviate $\tau v_1 \Rightarrow \tau v_2$ in C as $\tau v_1 \Rightarrow \tau v_2$ when C refers to the current closure under computation. We call $\tau v_1 \Rightarrow \tau v_2, \ldots, \tau v_n \Rightarrow \tau v_1$ a cycle, and we write $\tau v_1 \overset{*}{\Rightarrow} \tau v_n$ if there exists a sequence $\tau v_1 \Rightarrow \tau v_2, \ldots, \tau v_{n-1} \Rightarrow \tau v_n$.

Definition 52 (Terminating CPA): Terminating CPA is the instantiation of the framework obtained by defining `Poly` as:

$$\texttt{Poly}((\forall\, \overline{t}.\, t \rightarrow \tau \setminus C) <: t_1 \rightarrow t_2, \tau v') = \Theta, \text{where for each } \alpha \in \overline{t},$$
$$\Theta(\alpha) = \begin{cases} \alpha^{erase(\tau v')} & \text{if } erase(\tau v') \overset{*}{\Rightarrow} erase((\forall\, \overline{t}.\, t \rightarrow \tau \setminus C)) \\ \alpha^{\tau v'} & \text{otherwise} \end{cases}$$

The new algorithm differs from Idealized CPA in just one case: when a closure of type $(\forall\, \overline{t}.\, t \rightarrow \tau \setminus C)$ is applied to argument type $\tau v'$ and we have $erase(\tau v') \overset{*}{\Rightarrow} erase((\forall\, \overline{t}.\, t \rightarrow \tau \setminus C))$ in the current closure, then by the definition of $\Rightarrow$, there would be a cycle: $erase(\tau v') \overset{*}{\Rightarrow} erase((\forall\, \overline{t}.\, t \rightarrow \tau \setminus C)) \Rightarrow erase(\tau v')$. In this case, instead of renaming type variables in $\overline{t}$ as in Idealized CPA, they are renamed to a form only dependent on $erase(\tau v')$. In this way, even if $(\forall\, \overline{t}.\, t \rightarrow \tau \setminus C)$ applies to different types which are different instantiations of $erase(\tau v')$, there is only one contour generated for them. We will prove shortly that this will ensure termination of the closure computation.

Applying the algorithm to example E_3, suppose that, by the inference rule (Abs), function $(\lambda d \ldots)$ has type τ_d and function $(\lambda y \ldots)$ has type τ_y. Since $(\lambda d \ldots)$ lexically encloses $(\lambda y \ldots)$, we have $\tau_y \Rightarrow \tau_d$; and, since $(\lambda d \ldots)$ applies to closures of $(\lambda y \ldots)$, we also have $\tau_d \Rightarrow \tau_y$. Thus a cycle is detected, only two contours are generated for $(\lambda d \ldots)$, and the algorithm terminates.

Theorem 53 (Termination): The Terminating CPA analysis terminates for arbitrary programs.

Proof: Suppose not, *i.e.*, for some program, its Terminating CPA closure C contains a $\forall$ type τv_0 which has an infinite number of contours. Then, there must exist at least one τv_1 s.t. τv_0 takes as arguments an infinite number of instantiations of $erase(\tau v_1)$, and an infinite number of contours are generated for those applications. To have an infinite number of instantiations of $erase(\tau v_1)$, there must exist a $\forall$ type τv_2 s.t. τv_2 contains $erase(\tau v_1)$ as a sub-term, every new contour of τv_2 causes the generation of a new instantiation of $erase(\tau v_1)$, and τv_2 has an infinite number of contours. Repeating this process gives an infinite sequence $erase(\tau v_0), erase(\tau v_1), \ldots erase(\tau v_i) \ldots$ where for each i, τv_{2*i} has infinite number of contours when applying to instantiations of $erase(\tau v_{2*i+1})$, and $erase(\tau v_i) \Rightarrow erase(\tau v_{i+1})$. Since the program is finite, there are finitely many $erase(\tau v)$ and there must be a cycle in the sequence. Thus, there exists j s.t. $erase(\tau v_{2*j}) \Rightarrow erase(\tau v_{2*j+1}) \overset{*}{\Rightarrow} erase(\tau v_{2*j})$ and τv_{2*j} has an infinite number of contours for applying to instantiations of $erase(\tau v_{2*j+1})$. But, by the definition of `Poly` for Terminating CPA, this is impossible. □

A terminating Data-Adaptive CPA analysis can be similarly defined except that, besides cycles in the Flow Dependency relation, cycles in call-graph also need to be detected.

6 Conclusions

We have defined a polymorphic constrained type-based framework for polyvariant flow analysis. Some particular contributions include: showing how a type system with parametric polymorphism may be used to model polyvariance as well as data polymorphism; modeling nCFA and CPA in our framework; a refinement of CPA in the presence of data polymorphism; and, an approach to ensure the termination of CPA-style analyses.

Acknowledgements

We would like to thank the anonymous referees for their helpful comments.

References

[1] Ole Agesen. The cartesian product algorithm. In *Proceedings ECOOP'95*, volume 952 of *Lecture notes in Computer Science*, 1995.

[2] Ole Agesen. *Concrete Type Inference: Delivering Object-Oriented Applications*. PhD thesis, Stanford University, 1996. Available as Sun Labs Technical Report SMLI TR-96-52.

[3] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the International Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, 1993.

[4] Anindya Bannerjee. A modular, polyvariant, and type-based closure analysis. In *International Conference on Functional Programming*, 1997.

[5] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-98)*, volume 33, 10 of *ACM SIGPLAN Notices*, pages 183–200, New York, October 18–22 1998. ACM Press.

[6] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Sound polymorphic type inference for objects. In *OOPSLA '95*, pages 169–184, 1995.

[7] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Type inference for recursively constrained types and its application to OOP. In *Proceedings of the 1995 Mathematical Foundations of Programming Semantics Conference*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995. http://www.elsevier.nl/locate/entcs/volume1.html.

[8] Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-97)*, volume 32, 5 of *ACM SIGPLAN Notices*, pages 235–248, New York, June 15–18 1997. ACM Press.

[9] Cormac Flanagan. *Effective Static Debugging via Componential Set-Based Analysis*. PhD thesis, Rice University, 1997.

[10] David Grove, Greg DeFouw, Jerey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 1997.

[11] Trevor Jim. What are principal typings and what are they good for? In *Conference Record of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, 1996.

[12] Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *Conference Record of the Twenty-Second Annual ACM Symposium on Principles of Programming Languages*, pages 393–408, 1995.

[13] Flemming Nielson and Hanne Riis Nielson. Infinitary control flow analysis: A collecting semantics for closure analysis. In *Conference Record of the Twenty-Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 332–345, 1997.

[14] John Plevyak and Andrew Chien. Precise concrete type inference for object-oriented languages. In *Proceedings of the Ninth Annual ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 324–340, 1994.

[15] François Pottier. A framework for type inference with subtyping. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1) of *ACM SIGPLAN Notices*, pages 228–238. ACM, June 1999.

[16] Jens Palsberg and Christina Pavlopoulou. From polyvariant flow information to intersection and union types. In *Proceedings of 25th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98)*, pages 197–208, 1998.

[17] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie-Mellon University, 1991. Available as CMU Technical Report CMU-CS-91-145.

[18] J. B. Wells, Allyn Dimock, Robert Muller, and Franklyn Turbak. A typed intermediate language for flow-directed compilation. In *Theory and Practice of Software Development (TAPSOFT)*, number 1214 in Lecture notes in Computer Science. Springer-Verlag, 1997.

[19] Andrew K. Wright and Suresh Jagannathan. Polymorphic splitting: an effective polyvariant flow analysis. *ACM Transactions on Programming Languages and Systems*, 20(1):166–207, January 1998.

On Exceptions Versus Continuations in the Presence of State

Hayo Thielecke*

Department of Computer Science
Queen Mary and Westfield College,
University of London
London E1 4NS UK
ht@dcs.qmw.ac.uk

Abstract. We compare the expressive power of exceptions and continuations when added to a language with local state in the setting of operational semantics. Continuations are shown to be more expressive than exceptions because they can cause a function call to return more than once, whereas exceptions only allow discarding part of the calling context.

1 Introduction

Exceptions are part of nearly all modern programming languages, including mainstream ones like Java and C++. Continuations are present only in Scheme and the New Jersey dialect of ML, yet are much more intensely studied by theoreticians and logicians. The relationship between exceptions and continuations is not as widely understood as one would hope, partly because continuations, though in some sense canonical, are more powerful than would at first appear, and because the control aspect of exceptions can be obscured by intricacies of typing and syntax.

We have recently shown that exceptions and continuations, when added to a purely functional base language, cannot express each other [11]. That paper affords a comparison of, and contrast between, exceptions and continuations under controlled laboratory conditions, without any contamination from other effects so to speak. In this sequel paper we would like to complete the picture by comparing exceptions and continuations in the presence of state. It is known (and one could call it "folklore") that in the presence of storable procedures, exceptions can be implemented by storing a current handler continuation. It is also plausible that the more advanced uses of continuations cannot be done with exceptions, even if state is available too. Hence we would expect a hierarchy rather than incomparability in the stateful setting.

Formally, we compare expressiveness by means of contextual equivalence. For instance, we showed that $(\lambda x.pxx)M \simeq pMM$ is a contextual equivalence in a language with exceptions, whereas continuations can break it, so that exceptions

* Supported by EPSRC grant GR/L54639

G. Smolka (Ed.): ESOP/ETAPS 2000, LNCS 1782, pp. 397–411, 2000.

cannot macro-express continuations. Apart from the formal result, we would like to see the equivalences in the stateless setting of [11] as formalizing, at least to some extent, the distinction between the dynamic (exceptions) and the static (continuations) forms of control. The equivalences here give a different perspective, namely of how both forms of control alter the meaning of procedure call. With exceptions, a procedure call may discard part of its calling context; with continuations, a procedure call may return any number of times. It could be said that this distinction reflects the way that control manipulates the call stack: exceptions may erase portions of the stack; continuations may in addition copy them. However, we can make this distinction using only fairly high-level definitions of languages with exceptions and continuations, and a comparison of expressiveness. (Though ideally one would hope for a precise connection between the equivalences that hold for the various forms of control and the demands they put on storage allocation.)

The notion of expressiveness used here was already mentioned by Landin [6, 7], and formalized by Felleisen [3]. The reader should be warned that this notion of expressiveness is very different from the one used by Lillibridge [8]. Lillibridge was concerned with the typing of exceptions in ML, whereas we are concerned only with the actual jumping, that is, raising and handling exceptions, and invoking continuations, respectively. The typing of exceptions in ML "is totally independent of the facility which allows us to raise and handle these wrapped-up objects or packets" [1]. While the language for exceptions used here most closely resembles ML, we do not rely on typing, so that everything is also applicable to the `catch`/`throw` construct in LISP [14, 13], as it is essentially a spartan exception mechanism without handlers.

The remainder of the paper is organized as follows. The main constructs and their operational semantics are defined in Section 2. We first answer a question from [11], by showing that local exceptions are more powerful than global ones in Section 3. The main result of the paper is that continuations in the presence of state are more powerful than exceptions, which is proved in Section 4. Section 5 sketches how the result here could fit into a more systematic comparison between exceptions and continuations based on how often the current continuation can be used. Section 6 concludes.

2 The Languages and Their Operational Semantics

We extend the language used in the companion paper [11] with state by adopting the "state convention" from the Definition of Standard ML [9]. To avoid clutter, the store is elided in the rules unless specified otherwise. Formally a rule

$$\frac{M_1 \Downarrow P_1 \quad \ldots \quad M_n \Downarrow P_n}{M \Downarrow P}$$

is taken to be shorthand for a rule in which the state changes are propagated:

$$\frac{s_0 \vdash M_1 \Downarrow P_1, s_1 \quad \ldots \quad s_{n-1} \vdash M_n \Downarrow P_n, s_n}{s_0 \vdash M \Downarrow P, s_n}$$

Table 1. Natural semantics of the functional subset

$$\frac{P \Downarrow (\lambda x.\ P_1) \quad Q \Downarrow\ V \quad P_1[x := V] \Downarrow R}{(P\ Q) \Downarrow R} \qquad \frac{N \Downarrow n}{(\mathtt{succ}\ N) \Downarrow (n+1)}$$

$$\frac{N \Downarrow 0}{(\mathtt{pred}\ N) \Downarrow 0} \qquad \frac{N \Downarrow (n+1)}{(\mathtt{pred}\ N) \Downarrow n}$$

$$\frac{N \Downarrow 0 \qquad P_1 \Downarrow R}{(\mathtt{if0}\ N\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2) \Downarrow R} \qquad \frac{N \Downarrow (n+1) \qquad P_2 \Downarrow R}{(\mathtt{if0}\ N\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2) \Downarrow R}$$

$$\frac{}{V \Downarrow V} \qquad \frac{}{(\mathtt{rec}\ f(x).\ P) \Downarrow (\lambda x.\ P[f := (\mathtt{rec}\ f(x).\ P)])}$$

Table 2. Natural semantics of exceptions

$$\frac{N \Downarrow e \quad P \Downarrow V}{(\mathtt{raise}\ N\ P) \Downarrow (\mathtt{raise}\ e\ V)} \qquad \frac{N \Downarrow e \quad P \Downarrow V' \quad Q \Downarrow (\mathtt{raise}\ e'\ V'') \quad e \neq e'}{(\mathtt{handle}\ N\ P\ Q) \Downarrow (\mathtt{raise}\ e'\ V'')}$$

$$\frac{N \Downarrow V \quad P \Downarrow V' \quad Q \Downarrow V''}{(\mathtt{handle}\ N\ P\ Q) \Downarrow V''} \qquad \frac{N \Downarrow e \quad P \Downarrow V \quad Q \Downarrow (\mathtt{raise}\ e\ V') \quad (V\ V') \Downarrow R}{(\mathtt{handle}\ N\ P\ Q) \Downarrow R}$$

$$\frac{N \Downarrow (\mathtt{raise}\ e\ V)}{(op\ N) \Downarrow (\mathtt{raise}\ e\ V)} \qquad \frac{N \Downarrow (\mathtt{raise}\ e\ V)}{(\mathtt{if0}\ N\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2) \Downarrow (\mathtt{raise}\ e\ V)}$$

$$\frac{P \Downarrow (\mathtt{raise}\ e\ V)}{(P\ Q) \Downarrow (\mathtt{raise}\ e\ V)} \qquad \frac{P \Downarrow V \quad Q \Downarrow (\mathtt{raise}\ e\ V')}{(P\ Q) \Downarrow (\mathtt{raise}\ e\ V')}$$

$$\frac{N \Downarrow (\mathtt{raise}\ e\ V)}{(\mathtt{raise}\ N\ P) \Downarrow (\mathtt{raise}\ e\ V)} \qquad \frac{N \Downarrow V \quad P \Downarrow (\mathtt{raise}\ e\ V')}{(\mathtt{raise}\ N\ P) \Downarrow (\mathtt{raise}\ e\ V')}$$

$$\frac{N \Downarrow (\mathtt{raise}\ e'\ V)}{(\mathtt{handle}\ N\ P\ Q) \Downarrow (\mathtt{raise}\ e'\ V)} \qquad \frac{N \Downarrow V \quad P \Downarrow (\mathtt{raise}\ e'\ V')}{(\mathtt{handle}\ N\ P\ Q) \Downarrow (\mathtt{raise}\ e'\ V')}$$

Table 3. Natural semantics of state

$$\frac{s \vdash M \Downarrow a, s_1}{s \vdash (\mathtt{!}M) \Downarrow s_1(a), s_1} \qquad \frac{s \vdash M \Downarrow (\mathtt{raise}\ e\ V), s_1}{s \vdash (\mathtt{!}M) \Downarrow (\mathtt{raise}\ e\ V), s_1}$$

$$\frac{s \vdash M \Downarrow V, s_1 \qquad a \notin \mathsf{dom}(s_1)}{s \vdash (\mathtt{ref}\ M) \Downarrow a, s_1 + \{a \mapsto V\}} \qquad \frac{s \vdash M \Downarrow (\mathtt{raise}\ e\ V), s_1}{s \vdash (\mathtt{ref}\ M) \Downarrow (\mathtt{raise}\ e\ V), s_1}$$

$$\frac{s \vdash M \Downarrow a, s_1 \qquad s_1 \vdash N \Downarrow V, s_2}{s \vdash (M \mathtt{:=} N) \Downarrow V, s_2 + \{a \mapsto V\}}$$

$$\frac{s \vdash M \Downarrow (\mathtt{raise}\ e\ V), s_1}{s \vdash (M \mathtt{:=} N) \Downarrow (\mathtt{raise}\ e\ V), s_1} \qquad \frac{s \vdash M \Downarrow V', s_1 \qquad s_1 \vdash N \Downarrow (\mathtt{raise}\ e\ V), s_2}{s \vdash (M \mathtt{:=} N) \Downarrow (\mathtt{raise}\ e\ V), s_2}$$

Table 4. Evaluation-context semantics of continuations and state

$$
\begin{array}{rl}
V ::= & x \mid n \mid a \mid \lambda x.M \mid \texttt{rec}\ f(x).\ M \mid \#E \\
E ::= & [\cdot] \mid (E\ M) \mid (V\ E) \mid (\texttt{succ}\ E) \mid (\texttt{pred}\ E) \mid (\texttt{if0}\ E\ \texttt{then}\ M\ \texttt{else}\ M) \\
\mid & (\texttt{callcc}\ E) \mid (\texttt{throw}\ E\ M) \mid (\texttt{throw}\ V\ E) \\
\mid & (\texttt{ref}\ E) \mid (\texttt{!}\ E) \mid (E\texttt{:=}M) \mid (V\texttt{:=}E)
\end{array}
$$

$$
\begin{array}{lll}
s, E[(\lambda x.\ P)\ V] & \rightarrow s, E[P[x := V]] & \\
s, E[\texttt{succ}\ n] & \rightarrow s, E[n+1] & \\
s, E[\texttt{pred}\ 0] & \rightarrow s, E[0] & \\
s, E[\texttt{pred}\ (n+1)] & \rightarrow s, E[n] & \\
s, E[\texttt{if0}\ 0\ \texttt{then}\ M\ \texttt{else}\ N] & \rightarrow s, E[M] & \\
s, E[\texttt{if0}\ (n+1)\ \texttt{then}\ M\ \texttt{else}\ N] & \rightarrow s, E[N] & \\
s, E[\texttt{rec}\ f(x).\ M] & \rightarrow s, E[M[f := (\lambda x.\ (\texttt{rec}\ f(x).\ M)\ x)]] & \\
s, E[\texttt{callcc}\ (\lambda x.\ P)] & \rightarrow s, E[P[x := (\#E)]] & \\
s, E[\texttt{throw}\ (\#E')\ V] & \rightarrow s, E'[V] & \\
s, E[\texttt{ref}\ V] & \rightarrow s + \{a \mapsto V\}, E[a] & \text{where } a \notin \mathsf{dom}(s) \\
s, E[\texttt{!}\ a] & \rightarrow s, E[s(a)] & \\
s, E[a\texttt{:=}V] & \rightarrow s + \{a \mapsto V\}, E[V] &
\end{array}
$$

This version of exceptions (based on the "simple exceptions" of Gunter, Rémy and Riecke [5]) differs from those in ML in that exceptions are not constructors. The fact that exceptions in ML are constructors is relevant chiefly if one does *not* want to raise them, using `exn` only as a universal type. For our purposes, there is no real difference, up to an occasional η-expansion.

Definition 1. *We define the following languages:*

- *Let* λ_V+ *be defined by the operational semantics rules in Table 1.*
- *Let* λ_V+**exn** *be defined by the operational semantics rules in Tables 1 and 2.*
- *Let* λ_V+**state** *be defined by the rules in Table 3 and those in Table 1 subject to the state convention.*
- *Let* λ_V+**exn**+**state** *be defined by the rules in Table 3, as well as those in Tables 1 and 2 subject to the state convention.*

The rules for state are based on those in the Definition of Standard ML [9] (rules (99) and (100) on page 42), except that `ref`, `!` and `:=` are treated as special forms, rather than identifiers. A state is a partial function from addresses to values. For a term M, let $\mathsf{Addr}(M)$ be the set of addresses occurring in M. A program is a closed term P not containing any addresses, that is $\mathsf{Addr}(P) = \emptyset$.

We also need a language with continuations and state:

Definition 2. *Let* λ_V+**cont**+**state** *be the defined by the operational semantics in Table 4.*

The small-step operational semantics of λ_V+**cont**+**state** with evaluation contexts is in the style of Felleisen [12], with store added. Both addresses a and reified continuations $\#E$ are run-time entities that cannot appear in source programs.

Let a context C be a term with a hole not containing addresses.

Definition 3. *Two terms P and P' are* contextually equivalent*, $P \simeq P'$, iff for all contexts C, we have $\emptyset \vdash C[P] \Downarrow n, s$ for some integer n, iff $\emptyset \vdash C[P'] \Downarrow n, s'$.*

Contextual equivalence is defined analogously for the small-step semantics. However, in the small-step semantics we will be concerned with breaking equivalences, a strong version of which is the following:

Definition 4. *Two terms P and P'* can be separated *iff there is a context C such that: $\emptyset, C[P] \to^* s, n$ for some integer n, and $\emptyset, C[P'] \to^* s', n'$ with $n \neq n'$.*

(Again, the definition for big-step is analogous.)

Local definitions and sequencing are the usual syntactic sugar:

$$(\texttt{let}\ x = M\ \texttt{in}\ N) \equiv (\lambda x.N)\,M$$
$$(M;N) \equiv (\lambda x.N)\,M \qquad \text{where } x \text{ is not free in } N$$

3 Local Exceptions Are More Powerful than Global Ones

In this section, we show that even a small amount of state affects our comparison of continuations and exceptions. It may be surprising that local (that is, under a λ) declarations should have state in them, but local exception declarations generate new exception names (somewhat like `gensym` in LISP), and the equality test implicit in the exception handler is enough to make this observable.

Proposition 1. *There are terms that are contextually equivalent in the language with global exceptions λ_V+***exn***, but which can be separated if local exceptions are added.*

Proof. In λ_V+**exn**, we have a contextual equivalence

$$(\lambda x.pxx)\,M \simeq pMM$$

The proof of [11, Proposition 1] generalizes to the untyped setting. But local exceptions can break this equivalence: see Figure 1 for a separating context. □

From our perspective, we would maintain that the equivalence holds for the pure control aspect of exceptions, and is broken only because local exceptions are a somewhat hybrid notion with state in them.

Since all we need from local exceptions here is that one term evaluates to 1 and another to 2, we do not give a formal semantics for them, referring the reader to the Definition of Standard ML [9] (for a notation closer to the one used here, see also [5]).

```
fun single m p = let val y = m 0 in p y y end;

fun double m p = p (m 0) (m 0);

fun localnewexn d  =
let
        exception e
        fun r d = raise e
        fun h f x = ((f 0) handle e => x)
in
        fn q => q r h
end;

fun separate copier =
  (copier localnewexn)
    (fn q1 => fn q2 =>
      q1 (fn r1 => fn h1 =>
        q2 (fn r2 => fn h2 =>
          h1 (fn d => h2 r1 1) 2)));

separate single;
val it = 1 : int
separate double;
val it = 2 : int
```

Fig. 1. A separating context using local exceptions in Standard ML

The point in separating (Figure 1) is that each call of `localnewexn` generates a new exception. The handler in `h2` can only handle the exception raised from `r1` if `h2` and `r1` come from the same call of `localnewexn`, as they do in `separate single`, but not in `separate double`.

Local exceptions are relevant for us for two reasons: first, they make the equivalence for exceptions used in [11] inapplicable; second, they can to some extent approximate downward continuations. The example in Figure 1 does perhaps not witness expressive power in any intuitive sense. A more practical example may be the following: can one define a function `f` that passes to some unknown function `g` a function `h` that when called jumps back into `f` (assuming `h` is called before the call of `f` has terminated, because otherwise this would be beyond exceptions). With downward continuations, one can easily do that: in λ_V+**cont**+**state**, we would write `f` as $\lambda g.\mathtt{callcc}(\lambda k.g\,(\lambda x.\mathtt{throw}\,k\,x))$. Even such pedestrian control constructs as **goto** in ALGOL and `longjmp()` in C could do this. Yet with the simple version of exceptions we have in λ_V+**exn**, a handler in `g` may catch whatever exception `h` wanted to use to jump into `f`. With local exceptions however, `f` could declare a local exception for `h` to raise, which would thus be distinct from any that `g` could handle. On the other hand, language designers specifically chose to equip `g` so that it can intercept jumps from `h` to `f`: in ML even local exceptions can be handled by using a variable (or just a wildcard) pattern in the handler, while LISP provides `unwind-protect`.

4 Exceptions Cannot Make Functions Return Twice

Encodings of exceptions in terms of stored continuations have been known for some time, and can probably be regarded as folklore [5]; see also Reynolds's textbook [10]. It would still be worthwhile to analyze encodings of the various notions of exceptions in more detail. But the fact that such an encoding is possible, and that consequently continuations and state are *at least as* expressive as exceptions and state, will be treated as a known result here. We will strengthen it by showing that continuations in the presence of state are *strictly more* expressive than exceptions.

Define terms R_1 and R_2 in λ_V+**state** by

$$R_j \equiv \lambda z.((\lambda x.\lambda y.(z\,0;\ x\,\texttt{:=}\ \texttt{!}y;\ y\,\texttt{:=}j;\ \texttt{!}x))\,(\texttt{ref}\,0)\,(\texttt{ref}\,0))$$

Informally, the idea is that j is hidden inside R_j. As the variables x and y are local, the only way to observe j would be to run the assignments after the call to z twice, so that j is first moved into y, and then x, whose value is returned at the end. With exceptions, that is impossible.

The proof uses a variant of the technique used for exceptions in [11], extended to deal with the store. First we define a relation needed for the induction hypothesis:

Definition 5. *We define relations $\sim$ and $\sim_A$, where A is a set of addresses, as follows:*

- *on terms, let $\sim$ be the least congruence such that $M \sim M$ and $R_j \sim R_{j'}$ for any integers j and j';*
- *for stores, let $s \sim_A s'$ iff $A \subseteq \mathsf{dom}(s) = \mathsf{dom}(s')$ and for all $a \in A$, $s(a) \sim s'(a)$ and $\mathsf{Addr}(s(a)) \subseteq A$;*
- *for stores together with terms, let $s, M \sim_A s', M'$ iff $s \sim_A s'$ and $M \sim M'$, and also $\mathsf{Addr}(M) \subseteq A$.*

Intuitively, $s, M \sim_A s', M'$ implies that M in store s and M' in store s' are linked in lockstep; but the stores may differ in addresses outside A, which are inaccessible from M.

Lemma 1. *Assume $s, P \sim_A s', P'$ and $s \vdash P \Downarrow Q, s_1$. Then there exist a term Q', a store s'_1 and a set of addresses A_1 such that*

- *$s' \vdash P' \Downarrow Q', s'_1$;*
- *$s_1, Q \sim_{A_1} s'_1, Q'$;*
- *$A \subseteq A_1$ and $(\mathsf{dom}(s) \setminus A) \subseteq (\mathsf{dom}(s_1) \setminus A_1)$;*
- *for all addresses $a \in \mathsf{dom}(s) \setminus A$, the stores satisfy $s_1(a) = s(a)$ and $s'_1(a) = s'(a)$.*

Proof. Proof by induction on the derivation of $s \vdash P \Downarrow Q, s_1$. We assume $s, P \sim_A s', P'$ and proceed by cases on the last rule applied in the derivation.

Case $P \equiv MN$ and $s \vdash MN \Downarrow Q, s_4$. The last rule is

$$\frac{s \vdash M \Downarrow \lambda z.M_1, s_1 \qquad s_1 \vdash N \Downarrow\ V_2, s_2 \qquad s_2 \vdash M_1[z := V_2] \Downarrow Q, s_4}{s \vdash M\ N \Downarrow Q, s_4}$$

As $MN = P \sim P'$, P' must be of the form $M'N'$. By the induction hypothesis applied to $s \vdash M \Downarrow (\lambda z.M_1), s_1$, we have $s' \vdash M' \Downarrow (\lambda z.M_1'), s_1'$, with $s_1, \lambda z.M_1 \sim_{A_1} s_1', \lambda z.M_1'$.
There are two possible cases implied by $\lambda z.M_1 \sim \lambda z.M_1'$: either $M_1 \sim M_1'$; or $\lambda z.M_1 = R_j$ and $\lambda z.M_1' = R_{j'}$. In the first case, the claim follows by repeatedly applying the induction hypothesis. So suppose the second, that $\lambda z.M_1 = R_j$ and $\lambda z.M_1' = R_{j'}$. We apply the induction hypothesis, giving us $s_1', N' \Downarrow V_2', s_2'$ with $s_2, V_2 \sim_{A_2} s_2', V_2'$. Now

$$M_1[z := V_2] = (\lambda x.\lambda y.(V_2\, 0;\ x\texttt{:=}\ \texttt{!}y;\ y\texttt{:=}j;\ \texttt{!}x))\, (\texttt{ref}\, 0)\, (\texttt{ref}\, 0)$$

This term will allocate two new addresses, so let $a, b \notin \mathsf{dom}(s_2)$. Then $s_2 \vdash M_1[z := V_2] \Downarrow Q, s_4$ iff

$$s_2 + \{a \mapsto 0, b \mapsto 0\} \vdash V_2\, 0; a\texttt{:=}\,\texttt{!}b; b\texttt{:=}j;\, \texttt{!}a \Downarrow Q, s_4$$

There are two possible cases, depending on whether $V_2\, 0$ in store $s_2 + \{a \mapsto 0, b \mapsto 0\}$ raises an exception or not. First, suppose it does, that is,

$$s_2 + \{a \mapsto 0, b \mapsto 0\} \vdash V_2\, 0 \Downarrow \texttt{raise}\ e\ V_3, s_3 \tag{1}$$

As $s_2 + \{a \mapsto 0, b \mapsto 0\}, V_2\, 0 \sim_{A_2} s_2 + \{a \mapsto 0, b \mapsto 0\}, V_2'\, 0$, the induction hypothesis implies

$$s_2' + \{a \mapsto 0, b \mapsto 0\} \vdash V_2'\, 0 \Downarrow \texttt{raise}\ e\ V_3', s_3'$$

with $\texttt{raise}\ e\ V_3, s_3 \sim_{A_2} \texttt{raise}\ e\ V_3', s_3'$. The exception propagates, devouring the difference between j and j' in this call of R_j, more technically:

$$\cfrac{\cfrac{\cfrac{s_2 + \{a \mapsto 0, b \mapsto 0\} \vdash V_2\, 0 \Downarrow \texttt{raise}\ e\ V_3, s_3}{s_2 + \{a \mapsto 0, b \mapsto 0\} \vdash V_2\, 0; a\texttt{:=}\,\texttt{!}b \Downarrow \texttt{raise}\ e\ V_3, s_3}}{s_2 + \{a \mapsto 0, b \mapsto 0\} \vdash V_2\, 0; a\texttt{:=}\,\texttt{!}b; b\texttt{:=}j \Downarrow \texttt{raise}\ e\ V_3, s_3}}{s_2 + \{a \mapsto 0, b \mapsto 0\} \vdash V_2\, 0; a\texttt{:=}\,\texttt{!}b; b\texttt{:=}j;\, \texttt{!}a \Downarrow \texttt{raise}\ e\ V_3, s_3}$$

That is, $s_2 + \{a \mapsto 0, b \mapsto 0\} \vdash M_1[z := V_2] \Downarrow \texttt{raise}\ e\ V_3, s_3$, hence the whole call raises an exception

$$s_2 + \{a \mapsto 0, b \mapsto 0\} \vdash MN \Downarrow \texttt{raise}\ e\ V_3, s_3 \tag{2}$$

Analogously for V_2'. Letting $Q = \texttt{raise}\ e\ V_3$ and $s_4 = s_3$, we are done for this subcase. Now assume $V_2\, 0$ does not raise an exception, so that there is a value V_3 returned by the call:

$$s_2 + \{a \mapsto 0, b \mapsto 0\} \vdash V_2\, 0 \Downarrow V_3, s_3 \tag{3}$$

We apply the induction hypothesis to the call $V_2\,0$, relying on the fact that V_2 can only reach addresses in A_2, so that it cannot modify the newly allocated a and b:

$$s_2 + \{a \mapsto 0, b \mapsto 0\}, V_2\,0 \sim_{A_2} s'_2 + \{a \mapsto 0, b \mapsto 0\}, V'_2\,0$$

The induction hypothesis thus gives us $s'_2 + \{a \mapsto 0, b \mapsto 0\} \vdash V'_2\,0 \Downarrow V'_3, s'_3$, and $s_3, V_3 \sim_{A_3} s'_3, V'_3$. As $b \in \mathsf{dom}(s_2 + \{a \mapsto 0, b \mapsto 0\})$, but $b \notin A_2$, we have $s_3(b) = 0$, and $s'_3(b) = 0$, and also $b \notin A_3$. Putting the pieces together, we derive:

$$s_2 + \{a \mapsto 0, b \mapsto 0\} \vdash (V_2\,0;\ a\texttt{:=}\ \texttt{!}b;\ b\texttt{:=}j;\ \texttt{!}a) \Downarrow 0, s_3 + \{b \mapsto j\}$$

hence

$$s_2 + \{a \mapsto 0, b \mapsto 0\}, (\lambda x.\lambda y.(V_2\,0;\ x\texttt{:=}\ \texttt{!}y;\ y\texttt{:=}j;\ \texttt{!}x))\,(\texttt{ref}\,0)\,(\texttt{ref}\,0)$$
$$\Downarrow 0, s_3 + \{b \mapsto j\}$$

Analogously

$$s'_2 + \{a \mapsto 0, b \mapsto 0\} \vdash (V'_2\,0;\ a\texttt{:=}\ \texttt{!}b;\ b\texttt{:=}j';\ \texttt{!}a) \Downarrow 0, s'_3 + \{b \mapsto j'\}$$

hence

$$s'_2 + \{a \mapsto 0, b \mapsto 0\}, (\lambda x.\lambda y.(V'_2\,0;\ x\texttt{:=}\ \texttt{!}y;\ y\texttt{:=}j';\ \texttt{!}x))\,(\texttt{ref}\,0)\,(\texttt{ref}\,0)$$
$$\Downarrow 0, s'_3 + \{b \mapsto j'\}$$

Thus

$$s \vdash M N \Downarrow 0, s_3 + \{b \mapsto j\} \tag{4}$$

and $s' \vdash M'N' \Downarrow 0, s_3 + \{b \mapsto j'\}$ with $s_3 + \{b \mapsto j\}, 0 \sim_{A_3} s'_3 + \{b \mapsto j'\}, 0$, as required. This is the linchpin of the whole proof: b holds j or j', respectively; but that is of no consequence, because b, lying outside of A_3, is garbage.

Case $P \equiv \texttt{!}M$ and $s \vdash \texttt{!}M \Downarrow s_1(a), s_1$. Hence $s \vdash M \Downarrow a, s_1$. As $\texttt{!}M \sim P'$, P' must be of the form $\texttt{!}M'$ with $M \sim M'$. By the induction hypothesis, $s' \vdash M' \Downarrow Q', s'_1$ with $s_1, a \sim_{A_1} s'_1, Q'$, and $\mathsf{Addr}(s(a)) \subseteq A_1$. As this implies $a \sim Q'$, we have $a = Q'$, so that $s' \vdash M' \Downarrow a, s'_1$, which implies $s' \vdash \texttt{!}M' \Downarrow s'_1(a), s'_1$. As $a = \mathsf{Addr}(Q') \subseteq A_1$, $s_1(a) \sim s'_1(a)$. Thus $s_1, s_1(a) \sim_{A_1} s'_1, s'_1(a)$, as required.

Case $P \equiv \texttt{ref}\,M$ and $s \vdash \texttt{ref}\,M \Downarrow a, s_1 + \{a \mapsto V\}$. Hence $s \vdash M \Downarrow V, s_1$ with $a \notin \mathsf{dom}(s_1)$. As $\texttt{ref}\,M \sim P'$, P' must be of the form $\texttt{ref}\,M'$ with $M \sim M'$. By the induction hypothesis, $s' \vdash M' \Downarrow V', s'_1$, with $s_1, V \sim_{A_1} s'_1, V'$ where $A \subseteq A_1$. Thus $s' \vdash \texttt{ref}\,M' \Downarrow a, s'_1 + \{a \mapsto V'\}$. (We can pick the same a, because $a \notin \mathsf{dom}(s'_1) = \mathsf{dom}(s_1)$.) Thus, $s' \vdash \texttt{ref}\,M' \Downarrow a, s'_1 + \{a \mapsto V'\}$ with

$$s_1 + \{a \mapsto V\}, a \sim_{A_1 \cup \{a\}} s'_1 + \{a \mapsto V'\}, a$$

Furthermore, $A \subseteq A_1 \cup \{a\}$ and $\mathsf{dom}(s) \backslash A \subseteq \mathsf{dom}(s_1 + \{a \mapsto V\}) \backslash (A_1 \cup \{a\})$.

Case $P \equiv (M := N)$ and $s \vdash M := N \Downarrow V, s_2 + \{a \mapsto V\}$. Hence $s \vdash M \Downarrow a, s_1$ and $s_1 \vdash N \Downarrow V, s_2$. As $M := N \sim P'$, P' must be of the form $M' := N'$, with $M \sim M'$ and $N \sim N'$. Applying the induction hypothesis to $s \vdash M \Downarrow a, s_1$ gives us Q' and s_1' such that $s' \vdash M' \Downarrow Q', s_1'$ and $s_1, a \sim_{A_1} s_1', Q'$. So $a \sim Q'$, which means $Q' = a$. Applying the induction hypothesis to $s_1 \vdash N \Downarrow V, s_2$ and $s_1, N \sim_{A_1} s_1', N'$ gives us V' and s_2' such that $s_2, V \sim_{A_2} s_2', V'$. Thus $s' \vdash M' := N' \Downarrow V', s_2' + \{a \mapsto V'\}$ with

$$s_2 + \{a \mapsto V\}, V \sim_{A_2} s_2' + \{a \mapsto V'\}, V'$$

as required. Assume b is an address with $b \in \mathsf{dom}(s) \setminus A$. Then $b \notin A_2$, and $s_2(b) = s_1(b) = s(b)$. Because $a \in A_2$, we have $b \neq a$, so that the store $s_2 + \{a \mapsto V\}$ still maps b to $s(b)$.

Otherwise. The last rule in the derivation must be of the form

$$\frac{s \vdash P_1 \Downarrow Q_1, s_1 \quad \ldots \quad s_{n-1} \vdash P_n \Downarrow Q_n, s_n}{s \vdash P \Downarrow Q, s_n}$$

Observe that the P_i in the antecedents are the immediate subterms of the P in the conclusion, and that conversely the Q in the conclusion is assembled from subterms of P and some of the Q_i in the antecedents. Hence:

$$\begin{aligned} \mathsf{Addr}(P_1) \cup \ldots \cup \mathsf{Addr}(P_n) &\subseteq \mathsf{Addr}(P) \\ \mathsf{Addr}(Q) &\subseteq \mathsf{Addr}(P) \cup \mathsf{Addr}(Q_1) \cup \ldots \cup \mathsf{Addr}(Q_n) \end{aligned}$$

Because $s, P \sim_A s', P'$, we have $s \sim s'$ and $P \sim P'$. The case $P \equiv R_j$ is trivial; otherwise we have $P \sim P'$ due to congruence, so there are $P_1', \ldots, P_n'$ with $P_i \sim P_i'$. Now $s, P_1 \sim_A s', P_1'$ (because $\mathsf{Addr}(P_1) \subseteq \mathsf{Addr}(P) \subseteq A$). By the induction hypothesis, there exist Q_1', s_1' and A_1 such that $s_1, Q_1 \sim_{A_1} s_1', Q_1'$ and $A \subseteq A_1$. Hence $s_1, P_2 \sim_{A_1} s_1', P_2'$, so that we can apply the induction hypothesis again to $s_1 \vdash P_2 \Downarrow Q_2, s_2$, and so on for all the antecedents. Finally, let Q' be built up from the Q_i' in the same way as Q is built up from the Q_i. By congruence, we have $Q \sim Q'$. As $s_n \sim s_n'$ and $\mathsf{Addr}(Q) \subseteq A_n$, we have $s_n, Q \sim_{A_n} s_n', Q'$, as required. □

We have thus shown that terms containing R_1 and R_2, respectively, proceed in lockstep. This implies that the R_j are contextually equivalent:

Lemma 2. *R_1 and R_2 are contextually equivalent in λ_V***+exn+state**.

Proof. Let C be a context. Suppose $\emptyset \vdash C[R_1] \Downarrow n, s$ for some integer n. We need to show that $C[R_2]$ also reduces to n. First, note that because $\sim$ on terms is defined to be a congruence with $R_1 \sim R_2$, we have $C[R_1] \sim C[R_2]$. As neither of these terms contains any addresses, they are related in the empty store with respect to the empty set of addresses, that is $\emptyset, C[R_1] \sim_\emptyset \emptyset, C[R_2]$. By Lemma 1, we have $\emptyset \vdash C[R_2] \Downarrow Q', s'$, for some s', Q' and A such that $s, n \sim_A s', Q'$. This implies $n \sim Q'$, so that $n = Q'$. The argument for showing that $\emptyset \vdash C[R_2] \Downarrow n, s$ implies that $C[R_1]$ in the empty store also reduces to n is symmetric. □

```
fun R j z = (fn x => fn y => (z 0; x := !y; y := j; !x))(ref 0)(ref 0);

fun C Rj =
callcc(fn top =>
        let
            val c = ref 0
            val s = ref top
            val d = Rj (fn p => callcc(fn r => (s := r; 0)))

        in
            (c := !c + 1;
             if !c = 2 then d else throw (!s) 0)
        end);

C(R 1);
val it = 1 : int
C(R 2);
val it = 2 : int
```

Fig. 2. A separating context using continuations and state in SML/NJ

Note that the proof would still go through if we changed the notion of observation to termination, or if we restricted to the typed subset.

It remains to show that the two terms that are indistinguishable with exceptions and state can be separated with continuations and state. To separate, the argument to R_j should save its continuation, then restart that continuation once, so the assignments get evaluated twice, thereby assigning j to x, and thus making the concealed j visible to the context.

Lemma 3. *In* λ_V+**cont**+**state***,* R_1 *and* R_2 *can be separated: there is a context* $C[\cdot]$ *such that*

$$\emptyset, C[R_1] \rightarrow^* s_1, 1$$
$$\emptyset, C[R_2] \rightarrow^* s'_1, 2$$

This is actually strictly stronger than R_1 and R_2 not being contextually equivalent (and it is machine-checkable by evaluation). We omit the lengthy calculation here, but see Figure 2 for the separating context written in Standard ML of New Jersey. From Lemmas 2 and 3, we conclude our main result:

Proposition 2. *There are* λ_V+**state** *terms that are contextually equivalent in* λ_V+**exn**+**state***, but which can be separated in* λ_V+**cont**+**state***.*

Combined with the known encodings of exceptions in terms of continuations and state, Proposition 2 means that continuations in the presence of state are *strictly more* expressive than exceptions.

5 Exceptions Can Discard the Calling Context

We have established that continuations are more expressive than exceptions by showing how they affect functions calls: using continuations, a call can return more than once. In this section, we aim at an analogous result for showing how exceptions give rise to added power compared to a language without control: using exceptions, a function call may discard part of the calling context. To put it facetiously as a contest between a term and its context, in the previous section we concocted a calling context whose main ingredient

$$\ldots z\,0; x\,\texttt{:=}\,\texttt{!}y; y\,\texttt{:=}j; \,\texttt{!}x \ldots$$

was chosen such that something good (for separating) would happen if only the callee z could return twice. Now we need a calling context in which something bad happens if the callee returns at all. One such context is given by sequencing with divergence. (The callee could avoid ever returning to the divergence by diverging itself, but for separating that would defeat the purpose.) More formally, there are terms that are contextually equivalent in the language with state but no control, and which can be separated in the language with exceptions and state (in fact, in any language with control). Let Ω be the diverging term $((\texttt{rec}\ f(x).\ f\,x)\,0)$. The recursion construct is used here so that everything generalizes to the typed subset of λ_V+**state**; if we are only concerned with the untyped language, we could just as well put $\Omega = (\lambda x.xx)(\lambda x.xx)$. Analogously to Lemma 2, we have

Lemma 4. *$(M;\Omega)$ and $(N;\Omega)$ are contextually equivalent in λ_V+***state***.*

Proof. (Sketch) Let $\sim$ be the least congruence such that $M \sim M$ and $M;\Omega \sim N;\Omega$ for any M and N. Let $\sim$ be defined on states pointwise, and let $s, P \sim s', P'$ iff $s \sim s'$ and $P \sim P'$. As in Lemma 1, we need to show that if $s, P \sim s', P'$ and $s \vdash P \Downarrow Q, s_1$, there is a Q such that $s' \vdash P' \Downarrow Q', s'_1$ with $s_1, Q \sim s'_1, Q'$. The only non-trivial case if $P \equiv (M;\Omega)$ and $P' \equiv (N;\Omega)$. Suppose one of them reduces to integer. If we do not have control constructs, that can only be the case if Ω reduces to a value. But here is no V such that $s, \Omega \Downarrow V, s_1$. (For suppose they were: there would be a derivation of minimal height, which would have to contain a smaller one.) □

The proof is simpler than for exceptions because when we relate two terms $(M;\Omega)$ and $(N;\Omega)$ it does not matter what M and N do, or what storage they allocate, as the Ω prevents any observation.

Lemma 5. *$(M;\Omega)$ and $(N;\Omega)$ can be separated in λ_V+***exn+state***.*

Proof. Let

$$\begin{aligned} M &= \texttt{raise}\ e\ 1 \\ N &= \texttt{raise}\ e\ 2 \\ C &= \texttt{handle}\ e\ [\cdot]\ (\lambda x.x) \end{aligned}$$

Then we have $\emptyset \vdash C[M;\Omega] \Downarrow 1, \emptyset$ and $\emptyset \vdash C[N;\Omega] \Downarrow 2, \emptyset$ in λ_V+**exn+state**. □

Proposition 3. *There are two terms in λ_V+ that are contextually equivalent in λ_V+**state**, but which can be separated in λ_V+**exn**+**state**.*

So far we have used operational semantics and contextual equivalence as a kind of probe to observe what control constructs can and cannot do. The astute reader may however have begun to suspect what the preoccupation with discarding the current continuation, or using it more than once, is driving at. In the remainder of this section, we sketch how the earlier material fits in with *linearity* in the setting of continuation semantics.

It is evident that in the continuation semantics of a language without control operators the current continuation is used in a linear way. For the function type we have

$$[\![A \to B]\!] = ([\![B]\!] \to \mathsf{Ans}) \multimap ([\![A]\!] \to \mathsf{Ans})$$

In a language with `callcc`, the $\multimap$ would have to be replaced by a $\to$, because the current continuation could be discarded or copied. Domain-theoretically, the linear arrow $\multimap$ can be interpreted as strict function space. So in the case of $M; \Omega$, the meaning of a looping term is $[\![\Omega]\!] = \perp$, and because

$$[\![M]\!] : \mathsf{Env} \to ([\![B]\!] \to \mathsf{Ans}) \multimap \mathsf{Ans}$$

is strict in its continuation argument, it preserves $\perp$. So it is immediate that

$$[\![M; \Omega]\!] \ = \perp = \ [\![N; \Omega]\!]$$

Moreover, this argument is robust in the sense that it works the same in the presence of state. In the semantics of a language with state, expression continuations take the store as an additional argument, so that the meaning of M is now:

$$[\![M]\!] : \mathsf{Env} \to \mathsf{Store} \to ([\![B]\!] \to \mathsf{Store} \to \mathsf{Ans}) \multimap \mathsf{Ans}$$

This is still strict in its continuation argument, mapping the divergent continuation $\perp$ to $\perp$.

All this requires little more than linear typechecking of the CPS transform. What seems encouraging, however, is that exceptions begin to fit into the same framework. For a language with exceptions or dynamic `catch`, the continuation semantics passes a current handler continuation. Here the current continuation and the handler continuation *together* are subject to linearity (this linearity is joint work in progress with Peter O'Hearn and Uday Reddy, which may appear elsewhere). Assuming that all exceptions are injected into some type E (like `exn` in ML), the linearity is seen most clearly in the function type:

$$[\![A \to B]\!] = (([\![B]\!] \to \mathsf{Ans}) \& ([\![E]\!] \to \mathsf{Ans})) \multimap ([\![A]\!] \to \mathsf{Ans})$$

(Note that this linear use of non-linear continuations is quite different from "linear continuations" [4]). Again the linearity would remain the same if state were added to the continuations. The current continuation can be discarded in favour of the handler, but never used twice. Exceptions thus occupy a middle ground between no control operators (*linear* usage of the current continuation) and first-class continuations (*intuitionistic*, that is unrestricted, usage). For this reason we regard Lemma 2, which confirms that with exceptions no function call can return twice, as more than a random equivalence: it seems to point towards deeper structural properties of control made observable by the presence of state (in that the `ref` construct allowed us to "stamp" continuations uniquely, and then to count their usage with assignments).

6 Conclusions and Directions for Further Research

It is striking how sensitive the comparison between exceptions and continuations is to the chosen measure of expressiveness: in Lillibridge's terms, "exceptions are strictly more powerful" than continuations [8]; in terms of contextual equivalence and in the absence of state they are incomparable [11]; while in the presence of state, continuations are strictly more expressive than exceptions. The last of these is perhaps the least surprising because closest to programming intuition.

Each of these notions is to some extent brittle. For instance, comparisons of expressiveness based on the ability to encode recursion are inapplicable if the language under consideration already has recursion—and in the presence of state (including storable procedures) that is inevitable, as one can use Landin's technique of "tying a knot in the store" to define the "imperative **Y**-combinator". On the other hand, the technique of witnessing expressive power by breaking equivalences, while more widely applicable, is not completely robust either, if other effects are added to the language that already break the equivalence: compare Proposition 1 and [11, Proposition 1]. (Equivalences may be broken for uninteresting as well as interesting reasons.) Furthermore, while we would claim that Proposition 2 confirms and backs up programming intuition, it can hardly be said to *express* the difference between exceptions and continuations. A type system for the restricted (linear or affine) use of the current continuation would come much closer to achieving this. Ideally, such a linear typing for continuation-passing style together with typed equivalences of the target language should entail the equivalences considered here; we hope that our results will give such a unified treatment something to aim for. It has been suggested to us that "of course exceptions are weaker—they're on the stack". Some substance might conceivably be added to such statements if it could be shown that linearity in the use of continuations by dynamic control constructs is what allows control information to be stack-allocated (see also [2, 15]).

Acknowledgements

Thanks to Jon Riecke, Peter O'Hearn and Josh Berdine.

References

[1] Andrew Appel, David MacQueen, Robin Milner, and Mads Tofte. Unifying exceptions with constructors in Standard ML. Technical Report ECS LFCS 88 55, Laboratory for Foundations of Computer Science, University of Edinburgh, June 1988.

[2] Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. Representing control in the presence of one-shot continuations. *ACM SIGPLAN Notices*, 31(5):99–107, May 1996.

[3] Matthias Felleisen. On the expressive power of programming languages. In *Science of Computer Programming*, volume 17, pages 35–75, 1991.

[4] Andrzej Filinski. Linear continuations. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, 1992.

[5] Carl A. Gunter, Didier Rémy, and Jon G. Riecke. A generalization of exceptions and control in ML-like languages. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 12–23, La Jolla, California, June 25–28, 1995. ACM SIGPLAN/SIGARCH and IFIP WG2.8, ACM Press.

[6] Peter J. Landin. A generalization of jumps and labels. Report, UNIVAC Systems Programming Research, August 1965.

[7] Peter J. Landin. A generalization of jumps and labels. *Higher-Order and Symbolic Computation*, 11(2), 1998. Reprint of [6].

[8] Mark Lillibridge. Exceptions are strictly more powerful than Call/CC. Technical Report CMU-CS-95-178, Carnegie Mellon University, July 1995.

[9] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[10] John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

[11] Jon G. Riecke and Hayo Thielecke. Typed exceptions and continuations cannot macro-express each other. In Jĭrí Wiedermann, Peter van Emde Boas, and Mogens Nielsen, editors, *Procedings ICALP '99*, volume 1644 of *LNCS*, pages 635–644. Springer Verlag, 1999.

[12] Dorai Sitaram and Matthias Felleisen. Reasoning with continuations II: full abstraction for models of control. In M. Wand, editor, *Lisp and Functional Programming*. ACM, 1990.

[13] Guy L. Steele. *Common Lisp: the Language*. Digital Press, 1990.

[14] Guy L. Steele and Richard P. Gabriel. The evolution of Lisp. In Richard L. Wexelblat, editor, *Proceedings of the Conference on History of Programming Languages*, volume 28(3) of *ACM Sigplan Notices*, pages 231–270, New York, NY, USA, April 1993. ACM Press.

[15] Mitchell Wand and Dino P. Oliva. Proving the correctness of storage representations. In *1992 ACM Conferenc on Lisp and Functional Programming*, pages 151–160. ACM, ACM, August 1992.

Equational Reasoning for Linking with First-Class Primitive Modules

J. B. Wells[1]* and René Vestergaard[1]

Heriot-Watt University

Abstract. Modules and linking are usually formalized by encodings which use the λ-calculus, records (possibly dependent), and possibly some construct for recursion. In contrast, we introduce the m-calculus, a calculus where the primitive constructs are modules, linking, and the selection and hiding of module components. The m-calculus supports smooth encodings of software structuring tools such as functions (λ-calculus), records, objects (ς-calculus), and mutually recursive definitions. The m-calculus can also express widely varying kinds of module systems as used in languages like C, Haskell, and ML. We prove the m-calculus is confluent, thereby showing that equational reasoning via the m-calculus is sensible and well behaved.

1 Introduction

A long version of this paper [44] which contains full proofs, more details and explanations, and comparisons with more calculi (including the calculus of Ancona and Zucca [5]), is available at `http://www.cee.hw.ac.uk/~jbw/papers/`.

1.1 Support for Modules in Established Languages

All programming languages need support for modular programs. For languages like C, conventions outside the definition of the language provide this support. Each source file is compiled to an object ("`.o`") file which plays the role of the module. The namespace of modules is simply the file system and linking of modules is specified via extra-linguistic mechanisms such as makefiles. Connections are hard-wired to the component name rather than the module name: If module X uses module Y, modules Z and W supplying components with the same names as those of Y can be substituted for Y. There is a single global namespace for component names. Mutual dependencies between modules is possible, but there is no mechanism for black-box reuse of modules and no support for hierarchical structuring of modules within modules.

Languages like Ada [10], Modula-3 [26], and Haskell [1] support a kind of module which we will call *packages*. With packages, there is a flat namespace of modules; by convention module names correspond to filenames. Connections are hard-wired to module names: If module X uses module Y, then any replacement

* Supporting grants: EPSRC GR/L 36963, NSF CCR–9417382 and CCR–9806747.

G. Smolka (Ed.): ESOP/ETAPS 2000, LNCS 1782, pp. 412–428, 2000.

for Y must also be named Y and support at least the components used by X. As with C, mutual dependencies are supported but black-box reuse and hierarchical structuring are not.

The Standard ML language [36] has a very sophisticated module system which supports functions from modules to modules. There is again a namespace of modules, but modules can be nested hierarchically. Connections can be specified by components of module X referring to a previously defined module Y by name. Connections can also be specified by defining a *functor*, a function from modules to modules: If module X depends on a module named Y, then a functor F can be defined whose meaning is the function (λY.X). The functor F can be applied to other modules to yield new concrete modules. This provides flexibility in linking modules. Although ML supports black-box reuse and hierarchical structuring, mutually recursive modules are not allowed. (Current research is addressing this issue, e.g., [15].)

1.2 Reasonable Goals for a Module Formalism

The wide variety of existing module systems have evolved to satisfy a number of goals. We have designed a formal system, the m-calculus, for specifying and reasoning about the behavior of such module systems. In designing the m-calculus, we believed that it should satisfy as many of the following goals as possible:

Reuse without copying or modification: It should be possible (1) to use an individual module more than once in a program, (2) for each use of a module to be connected to other modules in different ways, and (3) for this to be done without changing or duplicating the source code of the module. This is called “black-box reuse” or *extensibility* [32]. Satisfying this requires that inter-module connections need not be specified inside the modules. We handle this in our m-calculus with *incomplete* (or *abstract*) modules and a *linking* operator.

Modules within the language: It should be possible to represent modules and linking together with the features of a core language in a single formalism. Reasoning about the behavior of real systems requires reasoning about all of the components of the real system simultaneously. Satisfying this goal requires either (1) that the module formalism should be able to represent core language features or (2) that it should be possible to combine the module formalism with formal systems for core languages. For our m-calculus we prefer approach (1) although approach (2) should be possible for many core languages.

First-class modules: It should be possible (1) for linking of modules to depend on arbitrary computations, (2) for modules to be created and loaded dynamically, (3) for modules to be passed as parameters and stored in data structures. This kind of power is necessary for reasoning about dynamic linking, a feature which is used in many C implementations on an *ad hoc* basis and is even appearing in language definitions such as that of Java [25]. Satisfying this requires either that the module formalism should support general computation or that it should be able to interact with the formalism used to represent the core language.

Closer fit to real systems: The module formalism should closely fit the actual features of real systems. For example, this means that the coding of modules and linking via λ-calculus, records, and a fix-point operator is inappropriate and cumbersome for languages with package-based module systems. This also means that the module formalism should have direct support for features of existing module systems such as mutual dependencies between modules as well as hierarchical structuring of modules. Our m-calculus easily models all three styles of module system that were described above. (Note that we do not deal with type issues in this paper.)

Sound and flexible equational reasoning: The module formalism should easily support (1) defining how a particular program will behave and (2) understanding the effect of program transformations. While many techniques have been developed for achieving (1), a particularly simple method is to define a reduction semantics, i.e., to define a set of evaluation contexts and a set of program-to-program rewrite rules. If this method is followed, (2) can be achieved by allowing the use of the rewrite rules in any context, not just in evaluation contexts, provided the consistency of the rules can be established. For our m-calculus, we establish internal consistency of the rewrite rules by proving the system is confluent.

1.3 A More General Notion of Module

The key to achieving the above-mentioned goals in the m-calculus is the use of a more general notion of module together with a linking operation. An incomplete or abstract module (introduced as a *mixin module* or a *mixin* in [4], formalized in a calculus in [5], and related to the notions of mixin in [17, 18, 13, 12]) is a collection of components of which some are exported (externally visible), some are private, and some are declared but not defined. We call the latter *deferred* components. For example, consider the following incomplete modules M_1 and M_2, where `N(f,g,i)` is an expression that depends on `f`, `g`, and `i` and similarly for `O(h)` and `P(f,i)`:

```
M1 = (module                        M2 = (module
        exported f = N(f,g,i)               deferred f
        deferred g                          exported g = P(f,i)
        deferred h                          deferred h
        private i = O(h))                   private i = Q)
```

Although the module components are named, the modules themselves do not bear names, i.e., they are anonymous, like abstractions in the λ-calculus [9]. In the m-calculus, we would write the above as:

$$\mathrm{M}_1 = \{\mathrm{f} \triangleright \mathrm{w} = \mathrm{N}(\mathrm{w},\mathrm{x},\mathrm{z}),\ \mathrm{g} \triangleright \mathrm{x} = \bullet,\ \mathrm{h} \triangleright \mathrm{y} = \bullet,\ \_ \triangleright \mathrm{z} = \mathrm{O}(\mathrm{y})\}$$
$$\mathrm{M}_2 = \{\mathrm{f} \triangleright \mathrm{w} = \bullet,\ \mathrm{g} \triangleright \mathrm{x} = \mathrm{P}(\mathrm{w},\mathrm{z}),\ \mathrm{h} \triangleright \mathrm{y} = \bullet,\ \_ \triangleright \mathrm{z} = \mathrm{Q}\}$$

In the m-calculus, each component has separate external and internal names from different namespaces (like in [27]). The internal names are subject to α-conversion and are necessary to support correctness of substitution in the m-calculus. The private components have only an internal name; the label “_”

means “no name”. Using standard m-calculus abbreviations, we can write the component $(\_ \triangleright \mathrm{z} = \mathrm{O}(\mathrm{y}))$ as simply $(\mathrm{z} = \mathrm{O}(\mathrm{y}))$. The component body “$\bullet$” indicates a deferred component where the body needs to be filled in by linking.

The meaning of deferred components is established by the linking operation. The result of the operation of linking M_1 and M_2, written $\mathrm{M}_1 \oplus \mathrm{M}_2$, is the new module M_3:

```
M3 = (module
        exported f = N(f,g,i)
        exported g = P(f,i')
        deferred h
        private i = O(h)
        private i' = Q)
```

In linking, deferred components are *concreted* by exported components of the other module. The two modules must not export components with the same name. Private components get renamed as necessary to avoid conflicts. Mutually recursive intermodule dependencies are supported — the example `f` and `g` components above depend on each other. In the m-calculus, M_3 is:

$$\mathrm{M}_3 = \{\mathrm{f} \triangleright \mathrm{w} = \mathrm{N}(\mathrm{w},\mathrm{x},\mathrm{z}),\ \mathrm{g} \triangleright \mathrm{x} = \mathrm{P}(\mathrm{w},\mathrm{z}'),\ \mathrm{h} \triangleright \mathrm{y} = \bullet,\ \_ \triangleright \mathrm{z} = \mathrm{O}(\mathrm{y}),\ \_ \triangleright \mathrm{z}' = \mathrm{Q}\}$$

The internal name of a component whose name does not match a component in the other module can be α-converted to a fresh name to avoid conflicts. The example does not illustrate this, but internal names of components with matching external names are α-converted to be the same to enable linking. In the m-calculus, M_3 being the result of $\mathrm{M}_1 \oplus \mathrm{M}_2$ is expressed by the single rewrite step $\mathrm{M}_1 \oplus \mathrm{M}_2 \longrightarrow \mathrm{M}_3$.

In addition to modules (which may be incomplete) and linking, only two other kinds of operations are needed for the m-calculus. One is selecting a component of a module, written $M.f$. The other needed operations are component hiding and sieving, written $M\backslash f$ and $M\backslash{-\mathcal{F}}$, necessary for certain kinds of namespace management. (There is also a “letrec” construct $\langle M \mid D \rangle$ which we could have chosen to encode as $\{f \triangleright x = M, D\}.f$.)

1.4 Contributions of This Paper

In section 2, we define the m-calculus, a calculus with modules and linking as primitive constructs. In the m-calculus modules are *first-class.* In section 3, we illustrate how various program construction mechanisms and module systems can be smoothly encoded in the m-calculus. In section 4, we give an overview of the proof of confluence, the bulk of which is treated in [44]. Confluence shows that equational reasoning via the m-calculus is sensible and well behaved and effectively means that rewriting is “meaning”-preserving. The m-calculus is the first calculus of linking for first-class primitive modules which has been proved confluent. (Modules are not first-class in [14, 35] and rewriting is not proven sound in [5].) In addition, in section 5, we discuss the related work.

As limitations, this paper does not deal with issues of types, strict evaluation, imperative effects, or classes and subclassing. As the λ-calculus serves for functions, the m-calculus serves as a theoretical foundation for examining the essence of modularity and linking. Analyses of further issues can be built on the m-calculus as they have been built on the λ-calculus.

1.5 Acknowledgements

We thank Zena Ariola and Lyn Turbak for inspirational discussions.

2 The m-Calculus

2.1 Syntax: Preterms and Raw Terms

The preterms of the m-calculus (the members of the set PreTerm) are given by the following grammar for M:

$$
\begin{array}{rcll}
w,x,y,z & \in & \mathrm{Var} & \text{(variables)} \\
f,g,h & \in & \mathrm{CompName} & \text{(component names)} \\
\mathcal{F} & \subseteq & \mathrm{CompName} & \text{(sets of component names)} \\
F & ::= & f \mid \_ & \text{(component label)} \\
B & ::= & M \mid \bullet & \text{(component body)} \\
c & ::= & (F \triangleright x = B) & \text{(component)} \\
D & ::= & c_1, \ldots, c_n \text{ where } n \geq 0 & \text{(component collection)} \\
M, N & ::= & \ x & \text{(variable)} \\
 & & \mid\ (M \backslash f) & \text{(component hiding)} \\
 & & \mid\ (M \backslash {-\mathcal{F}}) & \text{(component sieving)} \\
 & & \mid\ (M \oplus N) & \text{(linking)} \\
 & & \mid\ (M.f) & \text{(component selection)} \\
 & & \mid\ \{D\} & \text{(module)} \\
 & & \mid\ \langle M \mid D \rangle & \text{(letrec)}
\end{array}
$$

Let $<$ when used on component names be some strict total order. The following operations on components and component collections are defined. Given a component $c = (F \triangleright x = B)$, we define $\mathrm{Label}(c) = F$, $\mathrm{Name}(c) = \mathrm{Label}(c)$ if $\mathrm{Label}(c) \neq \_$ (otherwise undefined), $\mathrm{Binder}(c) = x$, and $\mathrm{Body}(c) = B$. Given a component collection $D = c_1, \ldots, c_n$, we define $|D| = n$, $D[i] = c_i$ if $1 \leq i \leq n$ and is otherwise undefined, $D[i := c] = c_1, \ldots, c_{i-1}, c, c_{i+1}, \ldots, c_n$ if $1 \leq i \leq n$ (otherwise undefined), $\mathrm{Names}(D) = \{\mathrm{Label}(c_1), \ldots, \mathrm{Label}(c_n)\} \setminus \{\_\}$, and $\mathrm{Binders}(D) = \{\mathrm{Binder}(c_1), \ldots, \mathrm{Binder}(c_n)\}$. Let $D[I] = D[i_1], \ldots, D[i_n]$ where $\{i_1, \ldots, i_n\} = I \cap \{1, \ldots, |D|\}$ and $i_1 < \ldots < i_n$. Let $D[\mathcal{F}] = D[i_1], \ldots, D[i_n]$ where $\{i_1, \ldots, i_n\} = \{\, i \mid \mathrm{Name}(D[i]) \in \mathcal{F} \,\}$ and $\mathrm{Name}(D[i_1]) < \ldots < \mathrm{Name}(D[i_n])$ ("components in D with names in $\mathcal{F}$"). Let $D[-\mathcal{F}] = D[\{\, i \mid \mathrm{Label}(D[i]) = F \notin \mathcal{F} \,\}]$ ("components in D with labels *not* in $\mathcal{F}$").

The following terminology is defined. Let $c = (F \triangleright x = B)$ be a component occurring at the top-level (not nested) in a collection D (i.e., $c = D[i]$ for some

i). If the label $F = \text{Label}(c)$ is a name f (and D belongs to a module), then c can be referred to by its name from outside the module for the purposes of linking, selection, or hiding. In this case we may call f an *external* name to distinguish it from the *binder* x which we may call an *internal* name. If F is the *anonymous marker*, written "_", then c is unnamed and is only accessible (internally) via its binder x. The variable $x = \text{Binder}(c)$ is a *binding occurrence* of x which binds free occurrences of x in the bodies of all of the components of D to the body of c. If D is the environment of a letrec $\langle M \mid D\rangle$, then the binder for x also binds free occurrences of x in M. Non-binding variable occurrences are *normal.* The body $B = \text{Body}(c)$ is either a preterm M or the *empty body*, written "•". The component c can be of four possible kinds, one of which will be forbidden:

– If $c = (f \triangleright x = M)$, then c is an *exported* or *output* component.
– If $c = (f \triangleright x = \bullet)$, then c is a *deferred* or *input* component.
– If $c = (\_ \triangleright x = M)$, then c is *private* or a *binding.*
– If $c = (\_ \triangleright x = \bullet)$, then this is an error (forbidden below).

A module with input components is *incomplete* or *abstract* and otherwise is *complete* or *concrete.*

The raw terms of the m-calculus (the members of RawTerm) are the preterms satisfying these conditions: (1) An unnamed component does not have an empty body. (2) Two named components in a collection do not have the same name. (3) Components in a collection bind distinct variables. (4) Components in a letrec environment are bindings (unnamed, non-empty bodies).

We use the following conventions for syntactic abbreviations. When writing a member of Term (cf. Section 2.3), a component $(F \triangleright x = B)$ may be written $(F \triangleright \_ = B)$ if no normal occurrences of x are bound by the component's binder. A component $(\_ \triangleright x = B)$ may be written as $(x = B)$; a component $(f \triangleright \_ = B)$ may be written $(f = B)$. The notation $M\backslash\{f_1, \ldots , f_n\}$ stands for $M\backslash f_1\backslash f_2 \cdots\backslash f_n$ where $f_1 < \cdots < f_n$. The expression (let $x = M$ in M') stands for $\langle M' \mid x = M\rangle$, provided $x \notin \text{FV}(M)$. Parentheses may be omitted; the possible ambiguities are resolved as by giving ".", "\", and "\−" higher precedence than "⊕" and making "⊕" left associative.

The free variables of a raw term are defined thus:

$$\begin{aligned}
\text{FV}(\bullet) &= \emptyset \qquad \text{FV}(x) = \{x\}\\
\text{FV}(M\backslash f) &= \text{FV}(M\backslash{-}\mathcal{F}) = \text{FV}(M.f) = \text{FV}(M)\\
\text{FV}(M_1 \oplus M_2) &= \text{FV}(M_1) \cup \text{FV}(M_2)\\
\text{FV}(\{D\}) &= \text{FV}(D) = (\textstyle\bigcup_{1\le i\le |D|} \text{FV}(\text{Body}(D[i]))) \setminus \text{Binders}(D)\\
\text{FV}(\langle M \mid D\rangle) &= (\text{FV}(M) \setminus \text{Binders}(D)) \cup \text{FV}(D)
\end{aligned}$$

The expression $\text{Capture}_x(M)$ denotes the set of bound variables in raw term M whose binding scope includes a free occurrence of the specific variable x. The operation $M[\![x := y]\!]$ renames to y all free occurrences of the variable x in M that are not in the scope of a binding of y.

A distinguished variable □, which is forbidden from being bound, is used as the *context hole.* A *context* is a raw term with one occurrence of □. Let C be a metavariable over contexts. The result of replacing the hole in C by the raw term M (*without* any variable renaming) is written $C[M]$.

2.2 Semantics: Structural and Computational Rewriting on Raw Terms

A rule "$X \rightsquigarrow Y$ if Z" is a schema which defines a *contraction* relation $\rightsquigarrow$ such that $M \rightsquigarrow N$ iff replacing the metavariables in X, Y, and Z by syntactic constructs of the appropriate sort yields, respectively, the terms M and N and a true proposition. A rule schema of the form $D \rightsquigarrow D'$ abbreviates the pair of rule schemas $\{D\} \rightsquigarrow \{D'\}$ and $\langle M \mid D\rangle \rightsquigarrow \langle M \mid D'\rangle$. If a rewrite relation $\longrightarrow$ is the *contextual closure* of a contraction relation $\rightsquigarrow$, this means that $\longrightarrow$ is the least relation such that $M \rightsquigarrow N$ implies $C[M] \longrightarrow C[N]$ for any context C.

The structural rewrite rules will use the following auxiliary definitions:

$$\begin{aligned}
&\mathrm{UnsafeNames}(x, D) = \textstyle\bigcup_{1 \le i \le |D|} \mathrm{Capture}_x(\mathrm{Body}(D[i])) \cup \mathrm{FV}(D) \cup \mathrm{Binders}(D)\\
&\mathrm{UnsafeNames}(x, \{D\}) = \mathrm{UnsafeNames}(x, D)\\
&\mathrm{UnsafeNames}(x, \langle M \mid D\rangle) = \mathrm{Capture}_x(M) \cup \mathrm{FV}(M) \cup \mathrm{UnsafeNames}(x, D)
\end{aligned}$$

$$\begin{aligned}
&\mathrm{BinderRenamed}(i, x, y, D, D')\\
&\iff \left(\begin{array}{l}
\quad\;\; D \;= (F_1 \rhd x_1 = B_1), \ldots ,(F_i \rhd x = B_i), \ldots ,(F_n \rhd x_n = B_n),\\
\text{and } D' = (F_1 \rhd x_1 = B_1'), \ldots ,(F_i \rhd y = B_i'), \ldots ,(F_n \rhd x_n = B_n'),\\
\text{and } B_j' = B_j[\![x := y]\!] \text{ for } 1 \le j \le n
\end{array}\right)
\end{aligned}$$

The structural rewrite rules are as follows:

$$\begin{array}{ll}
(\alpha\textbf{-letrec}) & \langle M \mid D\rangle \rightsquigarrow \langle M[\![x := y]\!] \mid D'\rangle\\
& \quad\text{if } \begin{cases} y \notin \mathrm{UnsafeNames}(x, \langle M \mid D\rangle),\\ \mathrm{BinderRenamed}(i, x, y, D, D') \end{cases}\\
(\alpha\textbf{-module}) & \{D\} \rightsquigarrow \{D'\}\\
& \quad\text{if } \begin{cases} y \notin \mathrm{UnsafeNames}(x, \{D\}),\\ \mathrm{BinderRenamed}(i, x, y, D, D') \end{cases}\\
(\textbf{comp-order}) & D_1, c_1, D_2, c_2, D_3 \rightsquigarrow D_1, c_2, D_2, c_1, D_3\\
(\textbf{link-commute}) & M_1 \oplus M_2 \rightsquigarrow M_2 \oplus M_1
\end{array}$$

The computational rewrite rules, which are presented in Figure 1, use the following auxiliary definitions. The expression $\mathrm{PickBody}(B, B')$ yields B if $B' = \bullet$, B' if $B = \bullet$, and is otherwise undefined. $\mathrm{DependsOn}_D$ is the least transitive, reflexive relation on $\{1, \ldots , |D|\}$ such that for all $i, j \in \{1, \ldots , |D|\}$,

$$\mathrm{DependsOn}_D(i, j) \Longleftarrow \left(\begin{array}{c} \mathrm{Binder}(D[j]) \in \mathrm{FV}(\mathrm{Body}(D[i]))\\ \text{or } (\mathrm{Body}(D[i]) = \bullet \text{ and } \mathrm{Label}(D[j]) \neq \_) \end{array}\right)$$

The structural and computational contraction relations, $\rightsquigarrow_{\mathrm{s}}$ and $\rightsquigarrow_{\mathrm{c}}$, are respectively the unions of the contraction relations of the structural and computational rules. The structural and computational rewrite relations, $\dashrightarrow_{\mathrm{s}}$ and

(**link**)
$$\{D\} \oplus \{D'\} \rightsquigarrow \{D[-\mathcal{F}], D'[-\mathcal{F}], D''\}$$
$$\text{if } \begin{cases} \mathcal{F} = \{f_1, \ldots, f_n\} = \mathrm{Names}(D) \cap \mathrm{Names}(D'), \\ \mathrm{Binders}(D[-\mathcal{F}]) \cap (\mathrm{Binders}(D') \cup \mathrm{FV}(D')) = \emptyset, \\ \mathrm{Binders}(D'[-\mathcal{F}]) \cap (\mathrm{Binders}(D) \cup \mathrm{FV}(D)) = \emptyset, \\ D[\mathcal{F}] \ = (f_1 \triangleright x_1 = B_1),\ \ldots\ ,(f_n \triangleright x_n = B_n), \\ D'[\mathcal{F}] = (f_1 \triangleright x_1 = B'_1),\ \ldots\ ,(f_n \triangleright x_n = B'_n), \\ D'' \quad = (f_1 \triangleright x_1 = B''_1),\ \ldots\ ,(f_n \triangleright x_n = B''_n), \\ B''_i = \mathrm{PickBody}(B_i, B'_i) \text{ is defined for } 1 \leq i \leq n \end{cases}$$

(**subst**)
$$D \rightsquigarrow D[i := (F_i \triangleright x_i = C[M_j])]$$
$$\text{if } \begin{cases} D[i] = (F_i \triangleright x_i = C[x_j]), \\ D[j] = (F_j \triangleright x_j = M_j), \\ \mathrm{Capture}_{\Box}(C) \cap (\{x_j\} \cup \mathrm{FV}(M_j)) = \emptyset, \\ \text{not } \mathrm{DependsOn}_D(j, i) \end{cases}$$

(**subst-letrec**)
$$\langle C[x] \mid D\rangle \rightsquigarrow \langle C[M] \mid D\rangle$$
$$\text{if } \begin{cases} D[i] = (\_ \triangleright x = M) \text{ for some } i, \\ \mathrm{Capture}_{\Box}(C) \cap (\{x\} \cup \mathrm{FV}(M)) = \emptyset \end{cases}$$

(**select**)
$$\{D\}.f \rightsquigarrow \langle x_i \mid D'\rangle$$
$$\text{if } \begin{cases} D \ = (F_1 \triangleright x_1 = M_1), \ldots\ ,(f \triangleright x_i = M_i), \ldots\ ,(F_n \triangleright x_n = M_n), \\ D' = (\_ \triangleright x_1 = M_1), \ldots\ , \ (\_ \triangleright x_i = M_i), \ldots\ , (\_ \triangleright x_n = M_n) \end{cases}$$

(**gc-module**)
$$\{D\} \rightsquigarrow \{D[I]\}$$
$$\text{if } \begin{cases} I, J \text{ partition } \{1,\ \ldots\ , |D|\}, \\ J \neq \emptyset, \\ \mathrm{Binders}(D[J]) \cap \mathrm{FV}(D[I]) = \emptyset, \\ \mathrm{Names}(D[J]) = \emptyset \end{cases}$$

(**gc-letrec**)
$$\langle M \mid D\rangle \rightsquigarrow \langle M \mid D[I]\rangle$$
$$\text{if } \begin{cases} I, J \text{ partition } \{1,\ \ldots\ , |D|\}, \\ J \neq \emptyset, \\ \mathrm{Binders}(D[J]) \cap (\mathrm{FV}(M) \cup \mathrm{FV}(D[I])) = \emptyset \end{cases}$$

(**empty-letrec**)
$$\langle M \mid \rangle \rightsquigarrow M$$

(**closure**)
$$\langle \{D\} \mid D'\rangle \rightsquigarrow \{D, D'\}$$
$$\text{if } \begin{cases} |D'| > 0, \\ \mathrm{Binders}(D) \cap (\mathrm{Binders}(D') \cup \mathrm{FV}(D')) = \emptyset \end{cases}$$

(**hide-present**)
$$\{D[i := (f \triangleright x = M)]\}\backslash f \rightsquigarrow \{D[i := (\_ \triangleright x = M)]\}$$

(**hide-absent**)
$$\{D\}\backslash f \rightsquigarrow \{D\}$$
$$\text{if } f \notin \mathrm{Names}(D)$$

(**sieve**)
$$\{D\}\backslash{-\mathcal{F}} \rightsquigarrow \{D'\}$$
$$\text{if } \begin{cases} D \ = (F_1 \triangleright x_1 = B_1), \ldots\ ,(F_n \triangleright x_n = B_n), \\ D' = (F'_1 \triangleright x_1 = B_1), \ldots\ ,(F'_n \triangleright x_n = B_n) \\ F'_i = \begin{cases} \_ \ \text{ if } F_i \notin \mathcal{F} \text{ and } B_i \neq \bullet \\ F_i \text{ if } F_i \in \mathcal{F} \end{cases} \qquad \text{for } 1 \leq i \leq n \end{cases}$$

Fig. 1. The computational rewrite rules.

$\dashrightarrow_{\mathrm{c}}$, are the contextual closures of $\rightsquigarrow_{\mathrm{s}}$ and $\rightsquigarrow_{\mathrm{c}}$, respectively. The structural equivalence relation, $=_{\mathrm{s}}$, is the transitive, reflexive, and symmetric closure of $\dashrightarrow_{\mathrm{s}}$. The (combined) contraction relation on raw terms is $\rightsquigarrow\ = \rightsquigarrow_{\mathrm{s}} \cup \rightsquigarrow_{\mathrm{c}}$ and the (combined) rewrite relation on raw terms is $\dashrightarrow\ = \dashrightarrow_{\mathrm{s}} \cup \dashrightarrow_{\mathrm{c}}$. The relations $\twoheadrightarrow_{\mathrm{s}}$, $\twoheadrightarrow_{\mathrm{c}}$, and $\twoheadrightarrow$ are the transitive, reflexive closures respectively of $\dashrightarrow_{\mathrm{s}}$, $\dashrightarrow_{\mathrm{c}}$, and $\dashrightarrow$.

While variables are subject to α-conversion, component names are not. This is similar to the way that a linker freely relocates (rename) offsets (internal names) within object files as necessary but does not generally rename symbol table entries (external names).

In the presence of cyclic bindings, the usual meta-level substitution and explicit substitution both result in size explosions and generally fail to provide the desired equations between programs. To avoid these difficulties, unlike the calculus of Ancona and Zucca [5], the m-calculus substitutes for one target at a time (via the (**subst**) and (**subst-letrec**) rules) in a style pioneered by Ariola, Blom, and Klop [8, 6, 7]. The m-calculus letrec contruct is, in a sense, a delayed substitution that allows avoiding duplication when a component is selected from a module.

The (**subst**) rule in Figure 1 uses the notion of one component of a collection *depending* on another to exclude certain rewriting possibilities. Without this condition of the (**subst**) rule, the m-calculus would not be confluent and would need a more complicated method as in [35] to prove soundness. Read $\mathrm{DependsOn}_D(j, i)$ as "component $D[j]$ depends on component $D[i]$ in collection D". The first condition of $\mathrm{DependsOn}_D$ handles syntactically evident dependencies. The second condition handles the possibility that a dependency will arise after linking the module $\{D\}$ with another module. Every input component is presumed to (potentially) depend on every output component, because there is always a module to link with that will cause the dependency to become real.

Most of the side conditions of the computational rules which concern the names of bound variables can be met by applying the structural rules first. This is the case for the use of Binders by (**link**) and (**closure**), the use of Capture by (**subst**) and (**subst-letrec**), and the way that (**link**) ensures that the binders of common components have the same name before linking. The side condition in (**closure**) that the component collection is non-empty merely avoids a trivial critical pair with (**empty-letrec**), making proofs easier.

The possible dynamic *errors* that can occur during computation in the m-calculus are (1) linking two modules whose output components are not disjoint, (2) selecting a component from an incomplete module, (3) selecting a component named f from a module which has no component named f, (4) hiding an input component, and (5) sieving out an input component. The following are examples of each of the kinds of errors:

$$\begin{array}{ll}
(1) & \{\mathrm{f} \triangleright \mathrm{w} = \bullet,\ \mathrm{g} \triangleright \mathrm{x} = \mathrm{M}\} \oplus \{\mathrm{f} \triangleright \mathrm{y} = \mathrm{N},\ \mathrm{g} \triangleright \mathrm{z} = \mathrm{N}'\} \\
(2) & \{\mathrm{f} \triangleright \mathrm{w} = \bullet,\ \mathrm{g} \triangleright \mathrm{x} = \mathrm{M}\}.\mathrm{g} \\
(3) & \{\mathrm{f} \triangleright \mathrm{w} = \mathrm{M},\ \mathrm{g} \triangleright \mathrm{x} = \mathrm{N}\}.\mathrm{h} \\
(4) & \{\mathrm{f} \triangleright \mathrm{w} = \bullet,\ \mathrm{g} \triangleright \mathrm{x} = \mathrm{N}\}\backslash\mathrm{f} \\
(5) & \{\mathrm{f} \triangleright \mathrm{w} = \bullet,\ \mathrm{g} \triangleright \mathrm{x} = \mathrm{N}\}\backslash{-}\{\mathrm{g}\}
\end{array}$$

2.3 The Calculus: Terms and Rewriting

The actual m-calculus is defined as $\mathcal{M} = (\mathrm{Term}, \longrightarrow) = (\mathrm{RawTerm}, \dashrightarrow_{\mathrm{c}})/ =_{\mathrm{s}}$. By this we mean that:

- The set Term of (real) terms is the set of equivalence classes of the raw terms under $=_{\mathrm{s}}$ (the structural equivalence relation).
- A term $[M]_{=_{\mathrm{s}}}$ (the equivalence class of raw term M under $=_{\mathrm{s}}$) rewrites to a term $[N]_{=_{\mathrm{s}}}$, written $[M]_{=_{\mathrm{s}}} \longrightarrow [N]_{=_{\mathrm{s}}}$, iff there are raw terms $M' \in [M]_{=_{\mathrm{s}}}$ and $N' \in [N]_{=_{\mathrm{s}}}$ such that $M' \dashrightarrow_{\mathrm{c}} N'$.

We assume throughout that raw terms are implicitly coerced to (real) terms when placed in a context requiring a term, e.g., $M \longrightarrow N$ means $[M]_{=_{\mathrm{s}}} \longrightarrow [N]_{=_{\mathrm{s}}}$. Let $\twoheadrightarrow$ be the transitive, reflexive closure of $\longrightarrow$.

3 Encoding Features in the m-Calculus

This section illustrates smooth encodings of various program construction mechanisms in the m-calculus.

3.1 Functions (λ-Calculus)

We define λ-calculus as syntactic sugar for m-calculus terms as follows, where "arg" and "res" are fixed component names (meaning "argument" and "result"):

$$\begin{array}{l}(\lambda x.M) = \{\mathrm{arg} \rhd x = \bullet,\ \mathrm{res} = M\} \\ (MM') = (M \oplus \{\mathrm{arg} = M'\}).\mathrm{res}\end{array}$$

This encoding is faithful to the meaning of the λ-calculus. We can verify the simulation of β-reduction as follows (where $M[x := M']$ is defined appropriately):

$$\begin{array}{lll}
 & & (\lambda x.M)M' \\
 & = & (\{\mathrm{arg} \rhd x = \bullet,\ \mathrm{res} = M\} \oplus \{\mathrm{arg} = M'\}).\mathrm{res} \\
 & = & (\{\mathrm{arg} \rhd x = \bullet,\ \mathrm{res} \rhd y = M\} \oplus \{\mathrm{arg} \rhd x = M'\}).\mathrm{res} \\
 & & \quad \text{where } y \notin \mathrm{FV}(M) \cup \mathrm{FV}(M') \text{ and } x \notin \mathrm{FV}(M') \\
(\textbf{link}) & \longrightarrow & \{\mathrm{arg} \rhd x = M',\ \mathrm{res} \rhd y = M\}.\mathrm{res} \\
(\textbf{select}) & \longrightarrow & \langle y \mid x = M', y = M\rangle \\
(\textbf{subst-letrec}) & \longrightarrow & \langle M \mid x = M', y = M\rangle \\
(\textbf{gc-letrec}) & \longrightarrow & \langle M \mid x = M'\rangle \\
(\textbf{subst-letrec}) & \twoheadrightarrow & \langle M[x := M'] \mid x = M'\rangle \\
(\textbf{gc-letrec}) & \longrightarrow & \langle M[x := M'] \mid \rangle \\
(\textbf{empty-letrec}) & \longrightarrow & M[x := M']
\end{array}$$

This encoding is similar to an independently developed encoding in [5]. It is only superficially related to the encoding of λ-calculus in ς-calculus [3].

3.2 Records and Record Operations

By the syntactic abbreviations defined in Section 2, record syntax is already accepted by the m-calculus. Furthermore, the expected rewrite rule for selection is simulated.

$$\{f_1 = M_1,\ \ldots,\ f_n = M_n\}.f_i \twoheadrightarrow M_i \quad \text{if } 1 \le i \le n$$

The simulation uses (**select**), (**gc-letrec**) (which can be applied because the internal names are not used), and (**empty-letrec**).

3.3 Objects (ς-Calculus)

The following record-of-methods encoding for the ς-calculus [3] works fine. We write "!" for the method invocation operator to avoid confusion with our component selection operator ".".

$$\begin{aligned}
[f_1 = \varsigma(x)M_1, \ldots, f_n = \varsigma(x)M_n] &= \{f_1 = \lambda x.M_1,\ \ldots,\ f_n = \lambda x.M_n\} \\
(M \Leftarrow f = \varsigma(x)M') &= M\backslash f \oplus [f = \varsigma(x)M'] \\
M!f &= (\text{let } x = M \text{ in } (x.f)x) \quad \text{where } x \text{ is fresh}
\end{aligned}$$

It is not hard to verify that the rewrite rules of the ς-calculus are simulated:

$$\begin{aligned}
&M!f_i \twoheadrightarrow M_i[x := M] \\
&\quad \text{where } M = [f_1 = \varsigma(x)M_1, \ldots, f_n = \varsigma(x)M_n] \text{ and } 1 \le i \le n \\
&[f_1 = \varsigma(x)M_1, \ldots, f_n = \varsigma(x)M_n] \Leftarrow f_i = \varsigma(x)M' \\
&\quad \twoheadrightarrow [f_1 = \varsigma(x)M_1, \ldots, f_i = \varsigma(x)M', \ldots, f_n = \varsigma(x)M_n] \quad \text{where } 1 \le i \le n
\end{aligned}$$

Of course, the real difficulty in dealing with objects is not in expressing their computational meaning but rather in devising the type system, an issue which we do not address in this paper.

3.4 Modules

C-style The m-calculus directly supports the modules of C-like languages. (The call-by-value evaluation and imperative features of C are left to future work.) Each object file O can be represented as a module M, and the linking of the modules $M_1, \ldots, M_n$ to form a program is represented as $P = (M_1 \oplus \ldots \oplus M_n)$. Invoking the program start routine is represented as (P.main).

Package-style For the package style of module system, a module named A which imports modules named $B_1, \ldots, B_n$ and exports entities named $f_1, \ldots, f_m$ is represented by an m-calculus module with one output component named A, and n input components named $B_1, \ldots, B_n$. The output component is in turn a module with n output components named $f_1, \ldots, f_m$ and some number of private components. The linking of modules $M_1, \ldots, M_n$ to form a program is again represented as $P = (M_1 \oplus \ldots \oplus M_n)$. Invoking the start routine of the program is now represented as (P.Main.main), i.e., there is a distinguished module named "Main" which must export a component named "main".

Consider for example the following Haskell program, where `P(A.f)` is an expression mentioning `A.f` and similarly for `Q(B.f,B.g)` and `N`:

```
module A (f) where
 f = N

module B (f, g) where
 import A
 g = P(A.f)
```

```
module Main (main) where
 import qualified B
 f = 5
 main = Q(B.f,B.g,f)
```

This program can be encoded in the m-calculus with these three modules, where A, B, main, and Main are component names:

$$\begin{aligned}
\mathrm{M_A} &= \{\mathrm{A} = \{\mathrm{f} = \mathrm{N}\}\}\\
\mathrm{M_B} &= \{\mathrm{A} \triangleright \mathrm{x} = \bullet,\ \mathrm{B} = \{\mathrm{g} = \mathrm{P}(\mathrm{x.f}),\ \mathrm{f} = \mathrm{x.f}\}\}\\
\mathrm{M_{Main}} &= \{\mathrm{B} \triangleright \mathrm{x} = \bullet,\ \mathrm{Main} = \{\mathrm{y} = 5,\ \mathrm{main} = \mathrm{Q}(\mathrm{x.f}, \mathrm{x.g}, \mathrm{y})\}\}
\end{aligned}$$

Note that the unexported "f" definition in Main is handled by a private component, so a variable "y" must be used instead of a component name. We can check the meaning of the program by rewriting:

$$\begin{aligned}
&(\mathrm{M_A} \oplus \mathrm{M_B} \oplus \mathrm{M_{Main}})\\
\twoheadrightarrow\ &\left\{\begin{matrix}\mathrm{A} \triangleright \mathrm{x} = \{\mathrm{f} = \mathrm{N}\},\ \mathrm{B} \triangleright \mathrm{z} = \{\mathrm{g} = \mathrm{P}(\mathrm{x.f}),\ \mathrm{f} = \mathrm{x.f}\},\\ \mathrm{Main} = \{\mathrm{y} = 5,\ \mathrm{main} = \mathrm{Q}(\mathrm{z.f}, \mathrm{z.g}, \mathrm{y})\}\end{matrix}\right\}\\
\twoheadrightarrow\ &\{\mathrm{A} = \{\mathrm{f} = \mathrm{N}\},\ \mathrm{B} = \{\mathrm{g} = \mathrm{P}(\mathrm{N}),\ \mathrm{f} = \mathrm{N}\},\ \mathrm{Main} = \{\mathrm{main} = \mathrm{Q}(\mathrm{N}, \mathrm{P}(\mathrm{N}), 5)\}\}
\end{aligned}$$

Thus, the overall meaning of the program is given by:

$$(\mathrm{M_A} \oplus \mathrm{M_B} \oplus \mathrm{M_{Main}}).\mathrm{Main.main} \twoheadrightarrow \mathrm{Q}(\mathrm{N}, \mathrm{P}(\mathrm{N}), 5)$$

In the Haskell example above, we used qualified names of the form `A.f`. In module `B` we could have used the unqualified name `f` to refer to the entity `A.f`. When a module imports more than one other module, a Haskell implementation uses its knowledge of the imported modules to determine the correct meaning of unqualified names. To encode Haskell modules into the m-calculus, we could use a translation that fully qualifies all names in each using information about the entire program.

However, it is desirable to reason about unqualified names in order to reason about modules separately. Consider for example the above Haskell program with module `B` replaced by the following modules:

```
module B (f, g, i) where
 import A
 import C
 i = 10
 g = P(f,h,i)
```

```
module C (h) where
 h = R
```

The name `f` in module `B` will end up referring to `A.f`, because there is no `C.f`, but this can not be determined without inspecting modules `A` and `C`. The name `i` in module `B` will only be legal if `A.i` and `C.i` do not exist. We can encode these modules as the following (extended) m-calculus modules:

$$\begin{aligned}
&\mathrm{M}'_{\mathrm{B}} = \left\{\begin{matrix}\mathrm{A} \triangleright \mathrm{y} = \bullet,\ \mathrm{C} \triangleright \mathrm{z} = \bullet,\ \mathrm{B} \triangleright \mathrm{w} = \{\mathrm{i} = 10,\ \mathrm{g} = \mathrm{P}(\mathrm{x.f}, \mathrm{x.h}, \mathrm{x.i}),\ \mathrm{f} = \mathrm{x.f}\}, \\ \mathrm{x} = (\mathrm{y} \setminus\!- \{\mathrm{f}, \mathrm{h}, \mathrm{i}\}) \oplus (\mathrm{z} \setminus\!- \{\mathrm{f}, \mathrm{h}, \mathrm{i}\}) \oplus (\mathrm{w} \setminus\!- \{\mathrm{f}, \mathrm{h}, \mathrm{i}\})\end{matrix}\right\} \\
&\mathrm{M}_{\mathrm{C}} = \{\mathrm{C} = \{\mathrm{h} = \mathrm{R}\}\}
\end{aligned}$$

The key idea of this encoding is adding the extra private component defining x to automatically resolve the unqualified names by picking them from whichever module is supplying them. Then we can verify that:

$$(\mathrm{M}_{\mathrm{A}} \oplus \mathrm{M}'_{\mathrm{B}} \oplus \mathrm{M}_{\mathrm{C}} \oplus \mathrm{M}_{\mathrm{Main}}).\mathrm{Main.main} \twoheadrightarrow \mathrm{Q}(\mathrm{N}, \mathrm{P}(\mathrm{N}, \mathrm{R}, 10), 5)$$

In the above example, observe that if M'_{B} is linked with two modules M'_{A} and M'_{C} whose A and C components both supply f, then the linking operation in M'_{B} which yields the private definition of x will get stuck. This corresponds to the fact that this is (usually) illegal in Haskell. (It *is* legal in Haskell for modules `B` and `C` to import module `A` and export `A.f`, and for module `D` to import both `B` and `C` and refer to the unqualified name `f`, because both `B.f` and `C.f` are aliases for `A.f`. It seems that the m-calculus would need to be extended to reason about sharing in order to encode this behavior.)

The Haskell module system has other features such as the ability to list which entities to import from a module, the ability to list entities *not* to import with unqualified names, local aliases for imported modules, and the ability to reexport all of the entities imported from another module. All of these features can be represented in the m-calculus.

ML-style The m-calculus can also represent the type-free aspects of ML-style modules. (The types, call-by-value evaluation, and imperative features of ML are left to future work.) Such module systems provide modules called *structures* as well as a λ-calculus (*functors* and *functor applications*) for manipulating them. A structure is essentially a dependent record; it is dependent in the sense that later fields can refer to the values of earlier fields. A functor is essentially a λ-abstraction whose body denotes a structure; a functor definition is the top-level binding of a functor to its name. ML structures can be encoded in the m-calculus as concrete modules. ML functors and functor applications can be encoded in the m-calculus via the λ-calculus encoding given in Section 3.1.

4 The Well-Behavedness of the Rewrite Rules

This section sketches the proof that the m-calculus is not only *confluent* but that it also satisfies the *finite developments* property. Due to space limitations, the details are only in the long version [44].

Proving these results uses a variation of the m-calculus which adds *redex marks* for tracking residuals of redexes of the computational rules and preventing contraction of freshly created redexes. Redexes of the (**link**), (**select**), (**empty-letrec**), (**closure**), (**hide-present**), (**hide-absent**), and (**sieve**) rules are marked at the root in the usual way. Redexes of (**subst**) and (**subst-letrec**) are marked at the variable which is the substitution target rather than at the root. Redexes of (**gc-module**) and (**gc-letrec**) are also not marked at the root; instead each component that can be garbage collected is marked. All marks are 0 except for substitution marks which must be 1 greater than all of the marks in the substitution source component body. (Due to the side condition on (**subst**) using DependsOn, it is always possible to mark all redexes in a term.)

Strong normalization (termination of rewriting) of the marked m-calculus is proved using a decreasing measure, the multiset of all marks in the term, in the well founded multiset ordering. Weak confluence of the marked m-calculus is proved by several lemmas established by careful case analyses together with a top-level proof structure that separately considers structural and computational rewrite steps. Our proof deals with and accounts for every structural operation (i.e., α-conversion and re-ordering) explicitly.

The combination of strong normalization and weak confluence of the marked m-calculus yields confluence of the marked m-calculus. Then developments are defined as those rewrite sequences of the m-calculus that can be lifted to the marked m-calculus. Using the confluence of the marked m-calculus, we prove that the results of any two coinitial developments can be joined by two further developments. Standard techniques then finish the proof of confluence of the m-calculus. Confluence is shown both for $\longrightarrow$ (on terms) and $\dashrightarrow$ (on raw terms).

5 Related Work

5.1 Calculi with Linking

Cardelli presents a simply-typed linking calculus for outermost-only modules without recursion [14]. Drossopoulou, Eisenbach, and Wragg give a module calculus for reasoning about the quirks of Java [16]. Ancona and Zucca give a calculus for linking modules which, although similar to ours, has a notion of substitution which we believe is less convenient and no published proof of rewriting properties [5]. Earlier, Ancona and Zucca also presented an algebra for simplifying module expressions which is not powerful enough to represent general computation [4]. Machkasova and Turbak give a calculus for linking outermost-only modules in a call-by-value language [35].

From a non-equational-reasoning point of view, Flatt and Felleisen give a calculus of modules with similar capabilities to ours [21]. Glew and Morrisett present a module calculus tailored towards dealing with linking of object files containing assembly-language-level code [24]. Waddell and Dybvig show how to encode modules and linking using Scheme's macro system [42].

5.2 Mixins

Duggan and Sourelis present a system of "mixin modules" which has the unique feature that when both modules have components with the same name, linking the modules results in a form of merging of the same-named components [17, 18]. Bracha and Lindstrom encode mixins using λ-calculus, records, and fix-point operators [13, 12]. Findler and Flatt describe using mixins and incomplete modules in actual programming [19]. Flatt and Krishnamurthi and Felleisen present a calculus with an operational semantics for mixins and classes in the context of Java [22].

5.3 Calculi for Cycles

Inspiring much of our formulation, Ariola and Klop did ground-breaking work on reasoning about λ-terms combined with a construct for mutually recursive definitions [8]. Ariola and Blom refined this work to prove consistency in the absence of confluence [6, 7].

5.4 ML-Style Modules vs. Types

Crary, Harper, and Puri describe how to extend the ML module system to deal with recursion [15]. Earlier work to add first-class modules (i.e., higher-order functors) to ML includes that of Russo [41], Harper and Lillibridge [27, 34], and Leroy [33]. Harper, Mitchell, and Moggi devised the *phase distinction* to show the decidability of type checking for the ML module system [28]. Jones shows how to avoid much of the complexity of typing ML-style modules via higher-order (parametric) signatures [31, 30].

5.5 Types vs. Concatenation and Extension for Records and Objects

When we extend our system with types, we will closely consider previous work on types for record concatenation [43, 29], extensible records [39, 23], and extensible objects [20, 40, 11].

References

[1] Haskell 98: A non-strict, purely functional language. Technical report, The Haskell 98 Committee, 1 Feb. 1999. Currently available at http://haskell.org.

[2] LNCS. Springer-Verlag, 2000.

[3] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[4] D. Ancona and E. Zucca. An algebra of mixin modules. In F. P. Presicce, editor, *Recent Trends in Algebraic Development Techniques (12th Int'l Workshop, WADT '97 — Selected Papers)*, number 1376 in LNCS, pages 92–106. Springer-Verlag, 1998.

[5] D. Ancona and E. Zucca. A primitive calculus for module systems. In G. Nadathur, editor, *Proc. Int'l Conf. on Principles and Practice of Declarative Programming*, LNCS, Paris, France, 29 Sept. – 1 Oct. 1999. Springer-Verlag.

[6] Z. M. Ariola and S. Blom. Cyclic lambda calculi. In *Theoretical Aspects Comput. Softw. : Int'l Conf.*, 1997.

[7] Z. M. Ariola and S. Blom. Lambda calculi plus letrec. Submitted, 3 July 1997.

[8] Z. M. Ariola and J. W. Klop. Lambda calculus with explicit recursion. *Inf. & Comput.*, 139:154–233, 1997.

[9] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, revised edition, 1984.

[10] J. G. P. Barnes. *Programming in Ada 95.* Addison-Wesley, 1996.

[11] V. Bono, M. Bugliesi, M. Dezani-Ciancaglini, and L. Liquori. Subtyping constraints for incomplete objects. *Fundamenta Informaticae*, 199X. To appear.

[12] G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity, and Multiple Inheritance.* PhD thesis, Univ. of Utah, Mar. 1992.

[13] G. Bracha and G. Lindstrom. Modularity meets inheritance. In *Proc. Int'l Conf. Computer Languages*, pages 282–290, 1992.

[14] L. Cardelli. Program fragments, linking, and modularization. In *Conf. Rec. POPL '97: 24th ACM Symp. Princ. of Prog. Langs.*, 1997.

[15] K. Crary, R. Harper, and S. Puri. What is a recursive module? In *Proc. ACM SIGPLAN '99 Conf. Prog. Lang. Design & Impl.*, 1997.

[16] S. Drossopoulou, S. Eisenbach, and D. Wragg. A fragment calculus — towards a model of separate compilation, linking and binary compatibility. In *Proc. 14th Ann. IEEE Symp. Logic in Computer Sci.*, July 1999.

[17] D. Duggan and C. Sourelis. Mixin modules. In *Proc. 1996 Int'l Conf. Functional Programming*, pages 262–273, 1996.

[18] D. Duggan and C. Sourelis. Parameterized modules, recursive modules, and mixin modules. In *ACM SIGPLAN Workshop on ML and its Applications*, 1998.

[19] R. B. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *Proc. 1998 Int'l Conf. Functional Programming*, 1998.

[20] K. Fisher, F. Honsell, and J. C. Mitchell. A lambda calculus of objects and method specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994.

[21] M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *Proc. ACM SIGPLAN '98 Conf. Prog. Lang. Design & Impl.*, 1998.

[22] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Conf. Rec. POPL '98: 25th ACM Symp. Princ. of Prog. Langs.*, 1998.

[23] B. R. Gaster and M. P. Jones. A polymorphic type system for extensible records and variants. Technical Report NOTTCS-TR-96-3, Univ. of Nottingham, 1996.

[24] N. Glew and G. Morrisett. Type-safe linking and modular assembly language. In POPL '99 [38].

[25] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison Wesley, 1996.

[26] S. P. Harbison. *Modula-3.* Prentice Hall, 1991.

[27] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In POPL '94 [37], pages 123–137.

[28] R. Harper, J. C. Mitchell, and E. Moggi. Higher-order modules and the phase distinction. In *Conf. Rec. 17th Ann. ACM Symp. Princ. of Prog. Langs.*, 1990.

[29] R. Harper and B. Pierce. A record calculus based on symmetric concatenation. Technical Report CMU-CS-90-157R, Carnegie Mellon Univ., 2 July 1991.

[30] M. P. Jones. From Hindley-Milner types to first-class structures. In *Proceedings of the Haskell Workshop*, La Jolla, California, U.S.A., 25 June 1995.

[31] M. P. Jones. Using parameterized signatures to express modular structure. In *Conf. Rec. POPL '96: 23rd ACM Symp. Princ. of Prog. Langs.*, 1996.

[32] S. Krishnamurthi and M. Felleisen. Toward a formal theory of extensible software. In *Sixth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Nov. 1998.

[33] X. Leroy. Manifest types, modules, and separate compilation. In POPL '94 [37], pages 109–122.

[34] M. Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems.* PhD thesis, Carnegie Mellon Univ., May 1997.

[35] E. Machkasova and F. Turbak. A calculus for link-time compilation. In *Proc. European Symp. on Programming* [2].

[36] R. Milner, M. Tofte, R. Harper, and D. B. MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1990.

[37] *Conf. Rec. 21st Ann. ACM Symp. Princ. of Prog. Langs.*, 1994.

[38] *Conf. Rec. POPL '99: 26th ACM Symp. Princ. of Prog. Langs.*, 1999.

[39] D. Rémy. Projective ML. In *Proc. 1992 ACM Conf. LISP Funct. Program.*, 1992.

[40] J. G. Riecke and C. A. Stone. Privacy via subsumption. *Theory and Practice of Object Systems*, 199X. To appear.

[41] C. V. Russo. *Types for Modules.* PhD thesis, Univ. of Edinburgh, 1998.

[42] O. Waddell and R. K. Dybvig. Extending the scope of syntactic abstraction. In POPL '99 [38].

[43] M. Wand. Type inference for record concatenation and multiple inheritance. In *Proc. 4th Ann. Symp. Logic in Computer Sci.*, pages 92–97, Pacific Grove, CA, U.S.A., June 5–8 1989. IEEE Comput. Soc. Press.

[44] J. B. Wells and R. Vestergaard. Confluent equational reasoning for linking with first-class primitive modules (long version). A short version is [45]. Full paper with three appendices for proofs, Aug. 1999.

[45] J. B. Wells and R. Vestergaard. Equational reasoning for linking with first-class primitive modules. In *Proc. European Symp. on Programming* [2]. A long version is [44].

Author Index