Programs as data objects: second symposium; proceedings / PADO 2001, Aarhus, Denmark, May 21 - 23, 2001. Olivier Danvy; Andrzej Filinski (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 2001

(Lecture notes in computer science ; Vol. 2053)
ISBN 3-540-42068-1


ISSN 0302-9743
ISBN 3-540-42068-1 Springer-Verlag Berlin Heidelberg New York

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

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de
© Springer-Verlag Berlin Heidelberg 2001
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Boller Medien design
Printed on acid-free paper SPIN: 10781527 06/3142 5 4 3 2 1 0
Preface

This volume constitutes the proceedings of the Second Symposium on Programs as Data Objects (PADO-II), held at the University of Aarhus, Denmark, on May 21–23, 2001. PADO-II was colocated with the Third International Workshop on Implicit Computational Complexity (ICC 2001) and the Seventeenth Conference on the Mathematical Foundations of Programming Semantics (MFPS XVII).

The first PADO was organized by Harald Ganzinger and Neil Jones, in 1985. This second symposium took place at the occasion of Neil Jones’s 60th birthday, and on his wish, we organized it as a research event. The call for papers was open and elicited 30 submissions from 12 countries. Overall, 145 reviews were collected, and based on these, the program committee selected 14 papers for presentation. With one exception, each submission received at least 4 reviews. Where relevant, a transcript of the (electronic) PC meeting was also enclosed.

PADO-II was sponsored by BRICS\(^1\) and the Esprit Working Group APPSEM, and organized in cooperation with the European Association for Programming Languages and Systems (EAPLS) and the Special Interest Group on Programming Languages of the Association for Computing Machinery (ACM SIGPLAN). We gratefully acknowledge their support.

We also extend our thanks to the PC members and external reviewers for their time and thoughts, Janne Kroun Christensen and Karen Kjær Møller for their organizational help, the \(<\text{bigwig}>\) project for hosting our submission website, and Daniel Damian for setting it up and maintaining it.

February 2001

Olivier Danvy and Andrzej Filinski

\(^{1}\) Basic Research in Computer Science (www.brics.dk), funded by the Danish National Research Foundation.
Conference Organization

Conference Co-chairs

Oliver Danvy, BRICS, University of Aarhus, Denmark
Andrzej Filinski, BRICS, University of Aarhus, Denmark

Program Committee

Torben Amtoft Boston University, USA
Charles Consel LaBRI / ENSERB, France
Catarina Coquand Chalmers University, Sweden
Radhia Cousot Ecole Polytechnique, France
Olivier Danvy University of Aarhus, Denmark
Andrzej Filinski University of Aarhus, Denmark
Yoshihiko Futamura Waseda University, Japan
Fritz Henglein IT University, Denmark
Peter Lee Carnegie Mellon University, USA
Y. Annie Liu SUNY Stony Brook, USA
Dave MacQueen Bell Labs, USA
James S. Royer Syracuse University, USA
Morten Heine Sørensen IT Practice, Denmark
Carolyn L. Talcott Stanford University, USA
Jon L White CommerceOne, Inc., USA

Additional Referees

Table of Contents

Invited Opening Talk

Program Analysis for Implicit Computational Complexity .................. 1
   Neil D. Jones

Contributed Papers

Deriving Pre-conditions for Array Bound Check Elimination .............. 2
   Wei-Ngan Chin, Siau-Cheng Khoo, Dana N. Xu

Type Systems for Useless-Variable Elimination ............................ 25
   Adam Fischbach, John Hannan

Boolean Constraints for Binding-Time Analysis ............................ 39
   Kevin Glynn, Peter J. Stuckey, Martin Sulzmann, Harald Søndergaard

Constraint Abstractions ................................................... 63
   Jörgen Gustavsson, Josef Svenningsson

Implementing Circularity Using Partial Evaluation .......................... 84
   Julia L. Lawall

Combining Forward and Backward Analyses of Temporal Properties ...... 103
   Damien Massé

Numerical Power Analysis .................................................. 117
   Isabella Mastroeni

Run-Time Bytecode Specialization .......................................... 138
   Hidehiko Masuhara, Akinori Yonezawa

A New Numerical Abstract Domain Based on Difference-Bound Matrices . 155
   Antoine Miné

Partial Evaluation for Class-Based Object-Oriented Languages .......... 173
   Ulrik P. Schultz

Driving in the Jungle ..................................................... 198
   Jens Peter Secher

Higher-Order Pattern Matching for Automatically Applying Fusion
Transformations ............................................................. 218
   Ganesh Sittampalam, Oege de Moor
Dynamic Partial Evaluation ........................................ 238
  Gregory T. Sullivan

Tag Elimination and Jones-Optimality ............................. 257
  Walid Taha, Henning Makholm, John Hughes

Invited Closing Talk

Bottom-Up Deduction with Deletion and Priorities .................. 276
  Harald Ganzinger

Author Index .......................................................... 279
This talk brings together ideas from two lines: automatic estimation of program running times, and implicit computational complexity. It describes ongoing research. Recent work in the two areas has been done by Bellanoni and Cook, Benzinger, Hofmann, Jones, Crary and Weirich, Leivant, Marion, Schwichtenberg, and others.

A main goal of implicit computational complexity is to “capture” complexity classes such as PTIMEF (polynomial-time computable functions) by computing formalisms that do not impose explicit bounds on time or space resources. Several researchers have succeeded in reaching this goal, a well-known example being the Bellanoni-Cook “safe primitive recursion on notation.”

It must be said, however, that recursion-theoretic formalisms such as primitive recursion are not very close to programming practice. In particular natural algorithms, as seen in introductory algorithm courses, often do not fall into existing implicit complexity classes. In some cases this has even been proven impossible, e.g., Colson established that primitive recursion alone cannot express computing the minimum of two numbers by the obvious linear-time algorithm.

In this work we identify a decidable class of algorithms such that all can be executed within polynomial time (or logarithmic space); and as well, the class includes many natural algorithms that are used in solving real problems.

For a standard first-order functional language we devise a type system giving information on the variations of its function parameters in terms of program inputs, and on run-time bounds for program-defined functions. Every syntactically correct program is well-typed, i.e., the language has a so-called “soft” type system.

The type information is extracted by data-flow analysis algorithms that extend the “size-change” framework of our POPL 2001 paper to account for running times as well as termination. The analysis allows automatic detection of programs that are guaranteed to run (or be runnable) in polynomial time.

Theorems are proven that this is indeed the case; and that the class is a proper generalization of “safe recursion” and some related schemes provided by other researchers. Several representative natural and efficient algorithms are seen to fall into the class, providing evidence that the class is “large enough.”
Deriving Pre-conditions for Array Bound Check Elimination

Wei-Ngan Chin, Siau-Cheng Khoo, and Dana N. Xu

School of Computing
National University of Singapore
{chinwn,khoosc,xuna}@comp.nus.edu.sg

Abstract. We present a high-level approach to array bound check optimization that is neither hampered by recursive functions, nor disabled by the presence of partially redundant checks. Our approach combines a forward analysis to infer precise contextual constraint at designated program points, and a backward method for deriving a safety pre-condition for each bound check. Both analyses are formulated with the help of a practical constraint solver based on Presburger formulae; resulting in an accurate and fully automatable optimization. The derived pre-conditions are also used to guide bound check specialization, for the purpose of eliminating partially redundant checks.

1 Introduction

Array bound check optimization has been extensively investigated over the last three decades [24, 12, 5, 17, 7, 8, 15], with renewed interests as recently as [3, 27, 23]. While successful bound check elimination can bring about measurable gains in performance, the importance of bound check optimization goes beyond these direct gains. In safety-oriented languages, such as Ada or Java, all bound violation must be faithfully reported through precise exception handling mechanism. With this, the presence of bound checks could potentially interfere with other program analyses. For example, data-flow based analysis must take into account potential loss in control flow should array bound violation occurs.

In this paper, we provide fresh insights into the problem of array bound check elimination, with the goal of coming up with a much more precise interprocedural optimization.

Let us first review the key problem of identifying bound checks for elimination. In general, under a given context, a check can be classified as either:

- unsafe;
- totally redundant;
- partially redundant.

A check is classified as unsafe if either a bound violation is expected to occur, or its safety condition is unknown. As a result, we cannot eliminate such a check. A check is classified as totally redundant if it can be proven that no
Deriving Pre-conditions for Array Bound Check Elimination

bound violation will occur. Lastly, a check is said to be partially redundant if we can identify a pre-condition that could ensure that the check becomes redundant.

Note that the classification of a check depends upon a given context. Specifically, a partially redundant check under a given context can become totally redundant when the context becomes “stronger”. (Contexts are expressed as predicate.)

To illustrate these three types of checks, consider the following two functions, expressed in a first-order functional language.

\[ \text{newsub}(arr, i, j) = \text{if } (0 \leq i \leq j) \text{ then } L1@H1@\text{sub}(arr, i) \text{ else } -1 \]

\[ \text{last}(arr) = \text{let } v = \text{length}(arr) \text{ in } L2@H2@\text{sub}(arr, v) \]

Arrays used in this paper are assumed to start at index 0, and can be accessed by primitive functions, such as \text{sub}. Furthermore, we annotate each array access \text{sub} call by some check labels, e.g. \( L1, H1 \), to identify the low and high bound checks respectively. The first function, \text{newsub}, accesses the element of an array after performing a test on its index parameter \( i \). For the access of \( \text{sub}(arr, i) \) to be safe, both a low bound check \( L1 = i \geq 0 \) and a high bound check \( H1 = i < \text{length}(arr) \) must be satisfied.

Under the context \( 0 \leq i \leq j \) of the if-branch, we can prove that \( L1 \) is totally redundant, but the same cannot be said about \( H1 \). In fact, the \( H1 \) check is partially redundant, and could be made redundant under appropriate pre-conditions, for e.g. \( j < \text{length}(arr) \).

The second function is meant to access the last element of a given array but it contains a bug. While the index of our array ranges from 0 to \( \text{length}(arr) - 1 \), this function used an index outside of this range. Hence, its upper bound check \( H2 = v < \text{length}(arr) \) is unsafe as it contradicts with the assertion \( v = \text{length}(arr) \) from the let statement.

Totally and partially redundant checks are traditionally identified by two separate techniques. As a matter of fact, forward data flow analysis[1, 15] which determines available expressions has been primarily used to identify totally redundant checks. An expression (or check) \( e \) is said to be available at program point \( p \) if some expression in an equivalence class of \( e \) has been computed on every path from entry to \( p \), and the constituent of \( e \) has not been redefined in the control flow graph (CFG). Using this information, the computation of an expression (or check) \( e \) at point \( p \) is redundant if \( e \) is available at that point.

Partially redundant checks are more difficult to handle. Traditionally, a backward dataflow analysis[8] is used to determine the anticipatability of expressions. An expression (or check) \( e \) is anticipatable at program point \( p \) if \( e \) is computed on every path from \( p \) to the exit of the CFG before any of its constituents are redefined. By hoisting an anticipatable expression to its safe earliest program point, selected checks can be made totally redundant. Historically, hoisting of anticipatable check is deemed as crucial for eliminating checks from loop-based

---

1 This includes the possibility that the check can be safely executed or it can be avoided.
programs. Unfortunately, hoisting of checks causes bound errors to be flagged at an earlier program point, creating problems for precise exception handling.

In this paper, we propose a new approach to eliminating array bound checks. Our approach begins with a forward contextual-constraint analysis that synthesize contexts for checks in a program. This is then followed by a backward derivation of weakest pre-conditions needed for checks to be eliminated.

For the example given above, our method determines that the lower bound check $L_1$ in the function $newsub$ is totally redundant; the upper bound check $H_2$ in the function $last$ is unsafe. Furthermore, the upper bound check $H_1$ of $newsub$ is determined to be partially redundant; the derived pre-condition being:

$$pre(H1) \equiv (i \leq -1) \lor (j < i \land 0 \leq i) \lor (i < length(arr))$$

To overcome the problem arising from hoisting of partially-redundant checks, we propose to use program specialization to selectively enforce contexts that are strong enough for eliminating partially-redundant checks. We note that such specialization technique is also advocated by [18] in their bound check optimization of Java programs.

Our new approach is built on top of an earlier work on sized-type inference [4], where we are able to automatically infer input/output size relation and also determine invariants for parameters of recursive functions over the sizes of data structures used. The inference is performed accurately and efficiently with the help of a constraint-solver on Presburger form [22]. The presence of sized type greatly enhances inter-procedural analysis of contextual constraints, which are crucial for identifying both unsafe and totally redundant checks. More importantly, accurate contextual constraint also helps in the derivation of safety pre-conditions for partially-redundant checks. With the derived pre-condition, we can provide specialized code to selectively eliminate partially-redundant checks based on the available contexts. The specialization process can be further tuned to provide a range of time/space tradeoff.

Our main contributions are:

1. To the best of our knowledge, we are the first to handle partially redundant checks through the backward derivation of safety pre-condition after contextual constraint has been gathered in a separate forward phase. This gives very accurate result for eliminating partially-redundant checks.
2. We deal directly with recursive functions, and the invariant synthesis is performed only once for each recursive function, instead of for every occurrence of checks within the function. Except for [26, 23] whose methods are restricted to totally redundant checks, almost all previous work for bounds check elimination deal with only loop-based programs.
3. We design a simple yet elegant approach to derive the weakest pre-condition (with respect to a given contextual constraint) for check elimination from the context of the check and the synthesized invariant. Our approach works seamlessly across recursive procedures.
4. We support inter-procedural optimization through backward propagation of a function’s pre-condition to its callers to become a check.
5. We introduce three forms of bound check specialization: \textit{polyvariant} for maximal specialization, \textit{monovariant} for minimal code duplication, and \textit{duovariant} specialization for a space/time tradeoff. While the idea of using context-based program specialization \cite{16, 6} is not new, our work is novel in its use of pre-condition for guiding effective specialization.

Section 2 gives an overview of our method by introducing sized types and the main steps towards bound check specialization. Section 3 formalizes the context synthesis as a forward analysis method. It also illustrates how invariants on recursive functions can be synthesized, so as to provide informative contexts for recursive functions. Section 4 describes the key steps for classifying checks, and the inter-procedural mechanism for deriving pre-conditions for each partially redundant check. Section 5 shows how the derived pre-conditions can be used to guide bounds check specialization; while Section 6 shows that the cost of analysis is within acceptable limit. Related work is compared in Section 7, before we discuss some future directions of research in the Section 8.

\begin{verbatim}
x \in \text{Var} \quad (\text{Variables}) 
a \in \text{Arr} \quad (\text{Array Names}) 
f \in \text{Name} \quad (\text{Function Names}) 
n \in \text{Int} \quad (\text{Integer Constants}) 
L \in \text{Label} \quad (\text{Labels for checks}) 
p \in \text{Prim} \quad (\text{Primitives}) 
\kappa \in \text{Call} \quad (\text{Calls}) 
e \in \text{Exp} \quad (\text{Expressions}) 
d \in \text{Def} \quad (\text{Function Definition})
\end{verbatim}

\textbf{Fig. 1. The Language Syntax}

2 Overview

We apply our technique to first-order typed functional language with strict semantics. Recursive functions in the language are confined to self-recursion. Currently, mutual recursion are encoded into self-recursion by appropriate tagging of input and output. The language is defined in Fig. 1. Note that the language syntax includes check labels (also called labels for brevity) that identify bound checks (ie., array bound checks or checks that originated from these bound checks). Check labels appears syntactically at calls to functions/operations that involve bound checks. However, We do not label self-recursive calls, as we provide slightly
different treatment to recursive function definitions (as explained in Section §3.2). Lastly, check labels are automatically inserted into programs by our analysis.

We restrict the arguments to a function to be just variables. This simplifies presentation, without loss of generality.

**Fig. 2. Syntax of Sized Types**

We only consider well-typed programs. We enhance the type system with the notion of sized types, which captures the size information about the underlying expressions/values. For a function, sized type reveals size relationships amongst the parameters and results of that function. The syntax of sized type is depicted in Fig. 2. It is a pair containing an annotated type and a Presburger formula. An annotated type expression augments an ordinary type expression with size variables; the relationship among these variables are expressed in the associated formula. In this paper, we consider only three basic types: Arrays, integers, and booleans. The annotated type for arrays is $\text{Arr}^v \tau$, where $v$ captures the array size; for integers, it is $\text{Int}^v$, where $v$ captures the integer value; for booleans, it is $\text{Bool}^v$, where $v$ can be either 0 or 1, representing the values False and True respectively. Occasionally, we omit writing size variables in the annotated type when these variables are unconstrained.

A sample program for our language is shown in Fig. 3. This program contains four functions that implement binary search. The main function $bsearch$ takes an array and a key in order to search for an element in the array. If found, the
getmid :: (Arr\(^a\) Int, Int\(^l\), Int\(^h\)) → (Int\(^m\), Int)
    Size \( a \geq 0 \land 2m \leq l + h \land l + h \leq 1 + 2m \)
getmid(arr, lo, hi) = let \( m = (lo + hi)/2 \)
    in let \( x = L3@H3@sub\) arr m in \((m, x)\)

\[
\begin{align*}
\text{cmp :: (Int}^i,\text{Int}^j) & \rightarrow \text{Int}^r \\
\text{Size} & (i < j \land r = -1) \lor (i = j \land r = 0) \lor (i > j \land r = 1) \\
\text{cmp}(k, x) & = \text{if } k < x \text{ then } -1 \text{ else if } k = x \text{ then } 0 \text{ else } 1 \\
\text{look :: (Arr}^a\text{ Int, Int}^l,\text{Int}^h,\text{Int}) & \rightarrow \text{Int}^r \\
\text{Size} & (a \geq 0) \land ((l \leq h) \lor (l > h \land r = 1)) \\
\text{Inv} & a^* = a \land l \leq h, l^* \land h^* \leq h \land \\
& 2 + 2l + 2h^* \leq h + 3l^* \land l + 2h^* < h + 2l^* \\
\text{look}(arr, lo, hi, key) & = \\
\text{if } (lo \leq hi) \text{ then } \\
& \text{let } (m, x) = L4@H4@getmid\) arr, lo, hi) \\
& \text{in let } t = \text{cmp}(key, x) \\
& \text{in if } t < 0 \text{ then } \text{look}(arr, lo, m - 1, key) \\
& \text{else if } (t == 0) \text{ then } m \text{ else } \text{look}(arr, m + 1, hi, key) \\
& \text{else } -1 \\
\text{bsearch :: (Arr}^a\text{ Int,Int}) & \rightarrow \text{Int} \\
\text{Size} & (a \geq 0) \\
\text{bsearch}(arr, key) & = \text{let } v = \text{length}(arr) \text{ in } L5@H5@look\) arr, 0, v - 1, key)
\end{align*}
\]

Fig. 3. Binary Search Program

corresponding array index is returned, otherwise \(-1\) is returned. The recursive invocation of binary search is carried out by the function \textit{look}.

2.1 Use of Sized Types

Sized type of a function captures the relationship between sizes of the function’s input and output. For instance, the annotated type for function \textit{cmp} is \((\text{Int}^i,\text{Int}^j) \rightarrow \text{Int}^r\), where \(i, j\) are the respective input values, and \(r\) is its output. The size constraint (identified by the keyword \textit{Size}) states three possible outputs for calling \textit{cmp}, depending on whether the argument \(k\) is less than, equal to, or greater than the argument \(x\).

Even more importantly, through sized-type inference [11], we can synthesize, for a recursive function, an invariant that describes changes in size of input arguments of the function during its nested recursive-call invocations. For example, an accurate invariant relationship between the (first three) argument sizes of any nested recursive calls to \textit{look}, \(a^*, l^*, h^*\), and the (first three) parameter sizes of the initial first call to \textit{look}, namely \(a, l, h\), has been captured as the following Presburger formula:

\[
\begin{align*}
\text{inv}(\text{look}) & = a^* = a \land l \leq h, l^* \land h^* \leq h \land \\
& 2 + 2l + 2h^* \leq h + 3l^* \land l + 2h^* < h + 2l^*
\end{align*}
\]
This invariant tells us that, in the successive recursive invocations of `look`, the size of the first argument remains unchanged. Also, the values of the argument `lo` (`\( l^* \)`) in the successive calls never get smaller than its initial value `l`, while those of `hi` (`\( h^* \)`) never get bigger than the initial value `h`. For example, if the first call to `look` is `look(2, 10)`, we know that successive recursive calls `look(lo, hi)` will satisfy the relationship `lo \geq 2 \land hi \leq 10`. Terminology-wise, we call the initial set of arguments/size variables (e.g., `a`, `l`, and `h`) the *initial arguments/sizes*, and that of an arbitrarily nested recursive call (e.g., `a^*, l^*,` and `h^*`) the *recursive arguments/sizes*.

\[
\begin{align*}
+ & :: \ (\text{Int}^i, \text{Int}^j) \rightarrow \text{Int}^k \\
\text{Size} & k = i + j \\
= & :: \ (\text{Int}^i, \text{Int}^j) \rightarrow \text{Bool}^b \\
\text{Size} & (0 \leq b \leq 1) \land ((i = j \land b = 1) \lor (i \neq j \land b = 0)) \\
\text{sub} & :: \ ((\text{Arr}^a \tau), \text{Int}^i) \rightarrow \tau \\
\text{update} & :: \ ((\text{Arr}^a \tau), \text{Int}^i, \tau) \rightarrow () \\
\text{Size} & a \geq 0 \\
\text{Req} & L : i \geq 0 \ ; \ H : i < a \\
\text{length} & :: \ (\text{Arr}^a \tau) \rightarrow \text{Int}^i \\
\text{newArr} & :: \ (\text{Int}^i, \tau) \rightarrow \text{Arr}^a \tau \\
\text{Size} & (i \geq 0 \land a = i)
\end{align*}
\]

**Fig. 4.** Sized Types of Some Primitives

Fig. 4 depicts the sized types of a collection of primitive functions used in the rest of this paper. For array-access operations (`sub` and `update`), we also include their respective pre-conditions, which must be satisfied for the operations to be safe. These pre-conditions are identified by the keyword `Req`.

Once the sized types of related functions have been inferred, we proceed to handle bound check optimization. The process works in a bottom-up fashion, starting with functions at the bottom of the call hierarchy. We list the steps involved below. Throughout the rest of the paper, we use the binary search program depicted in Fig. 3 as the running example.

1. **Step 1** Forward contextual-constraint analysis.
2. **Step 2** Backward pre-condition derivation.
3. **Step 3** Bound check specialization.

These steps are described in details in the following sections.

### 3 Context Synthesis

We begin by determining the context within which a check occurs. Contextual information is described in Presburger form. It is called *contextual constraint*,
and is gathered by traversing the syntax tree of the function body, beginning from the root of the tree to a check-labelled call. Constraints gathered during traversal include constraint for selecting a branch (of `if`-expression), assertion about the sizes of local variables, and post-conditions of function calls.

![Fig. 5. Definition of the Context-Derivation Function](C)
During the traversal of the syntax tree, $\mathcal{C}$ updates $\Gamma$ with sized types of locally defined variables. It also maintains a constraint $\psi$ that captures the context of the subject expression. Initially, $\psi$ is set to the value $True$. When a branch of an if-expression is chosen, the constraint leading to this decision is captured in $\psi$. When a labelled call is encountered, its contextual constraint is derived by combining (via conjunction) $\psi$ with related constraints kept in the environment $\rho$. The result is expressed as $\mathcal{F}_{\Gamma,\psi}$. Formally, it is defined as follows:

$$\mathcal{F}_{\Gamma,\psi} = \land \left( \bigcup_{i \geq 0} \Phi_i \right)$$

where

$$\Phi_0 = \{ \psi \}$$

$$\Phi_{i+1} = \{ \phi \mid \exists x, \tau. \Gamma \llbracket x \rrbracket = (\tau, \phi); \text{fv}(\phi) \cap \text{fv}(\Phi_i) \neq \emptyset; \phi \notin \bigcup_{j \leq i} \Phi_j \}$$

As the environment $\Gamma$ is finite, computation of $\mathcal{F}_{\Gamma,\phi}$ always terminates.

Notation-wise, in Fig. 5, function $\text{newVar}$ returns a new variable. Function $\alpha$ performs renaming of size variables (like $\alpha$-conversion). It is overloaded so that it can take in either an annotated type or a sized type (which is a pair). It consistently renames all size variables occurring in its argument. Lastly, operation $\text{eq } \tau_1 \tau_2$ produces a conjunction that equates the corresponding size variables of two annotated types. For instance, $(\text{eq } \text{Int}^a \text{Int}^w)$ produces the constraint $(v = w)$. Lastly, $\Gamma[x :: (\tau, \phi)]$ denotes updating of the environment $\Gamma$ by a new binding of $x$ to a sized type $(\tau, \phi)$.

As an example, for the following function definition,

\[
\text{newsu}b :: (\text{Arr}^a \text{Int}, \text{Int}^i, \text{Int}^j) \to \text{Int}
\]

\[
\text{newsu}b(arr, i, j) = \text{if } 0 \leq i \leq j \text{ then } L1@H1@\text{sub}(arr, i) \text{ else } -1
\]

$\mathcal{C}$ determines the context for the labelled call to be:

$$\text{ctx}(L1) = \text{ctx}(H1) = a \geq 0 \land 0 \leq i \leq j$$

### 3.1 Recursive-Call-Invariant Synthesis

Invariant synthesis is in general a hard problem for recursive function definitions. Two pieces of invariant information are useful. First, invariant describing input/output size relation (ie., its sized type) of a recursive function can be propagated across functions to achieve better context synthesis. Computing such invariant is not always possible, however, as sometimes the precise relation between input- and output-size is beyond Presburger formulation.

Second, recursive-call invariant captures the argument-size relationship between an initial call and an arbitrarily nested recursive call of the same function. This relation is needed for synthesizing the contextual constraint of a labelled call invoked at arbitrarily nested depth. Fortunately, such relation can often be formulated precisely using Presburger formula.

Computation of recursive-call invariant proceeds as follows: We first compute the constraint relating the parameter sizes of a function and the argument sizes of all recursive calls textually occurring in the function body. Conceptually,
this constraint spells out the change in argument size during **one unfolding** of the recursive call. It can be captured by a procedure similar to the context computation \( C \). For example, consider the function \( \text{look} \) defined in the binary search program. We obtain the following constraint, which we call \( U \):

\[
U := [a, l, h] \rightarrow [a^*, l^*, h^*] : h \geq l \land \\
\exists (m \cdot 2m \leq h + l \land h + l \leq 1 + 2m \land \\
((l^* = l \land h^* = m - 1) \lor (l^* = m + 1 \land h^* = h)))
\]

The notation used here is adapted from literature work in Omega Calculator \[22\]. It specifies the constraint between two sets of variables: \([a, l, h]\) and \([a^*, l^*, h^*]\). The former is the set of initial parameter sizes (they are called the **source**), and the latter being the set of argument sizes of a recursive call (they are called the **target**).

Next, we perform inductive computation to infer the change in argument size resulting from arbitrary number of recursive-call unfolding. The result is the recursive-call invariant. In the case of \( \text{look} \), we have:

\[
\text{inv}(\text{look}) = a^* = a \land l \leq h, l^* \land h^* \leq h \land \\
2 + 2l + 2h^* \leq h + 3l^* \land l + 2h^* < h + 2l^*
\]

Several researchers, including the present authors, have proposed different techniques for synthesizing invariants. As these techniques are complementary in power and efficiency, we believe a collection of these techniques is needed to do a decent job. This includes:

**Polyhedra analysis.** This is proposed and developed by Cousot and Halbwachs \[5, 11, 10\], as well as King and his co-workers \[2, 14\]. It is an abstract interpretation approach to finding the input/output size relation through fixed-point computation over linear constraint. Both convex-hull operation (to eliminate multiple disjuncts) and widening operation (to generalize a constraint by dropping some conjuncts that cannot be subsumed by others in an ascending chain of constraints) are used as generalization techniques to ensure termination of the analysis. For recursive-call invariant computation, we modify this analysis by ignoring the degenerated case of a recursive definition from our computation. As an example, Fig. 6 illustrates a trace of such computation for the function \( \text{look} \) with the aid of the Omega calculator.

In the above, lines begin with \# are comments; lines end with ; are commands to the Omega Calculator \[22\]; outputs from the Calculator are indented rightward. \((\text{hull } U)\) computes the convex hull of \( U \) (viewed as a relation) and \( \text{widen}(U_2, U_3) \) generalizes \( U_2 \) to yield a constraint \( W_2 \) such that both \( U_2 \) and \( U_3 \) are instances of \( W_2 \). (We refer the reader to the work of Halbwachs \[9, 10\] for detail description of these two operations.) \( \text{union} \) signifies disjunction, \( \text{compose} \) combines two constraints by matching (and eliminating) the target of the former with the source of the latter. The second and third steps of the above trace above are iterative computation of fixed-point computation. The last command checks if a fixed-point is reached.

**Transitive-closure operation.** This is a fixed-point operation provided in the Omega Calculator \[22\]. Given a linear constraint expressed in the form of
Fig. 6. A trace of Omega Calculation of Recursive-call Invariant relation (such as $U$ above), the transitive-closure operation aims to compute its least fixed point. A least fixed-point of $U$ is defined as $\bigvee_{i>0} (U^i)$, where $U^1 = U$, and $U^{i+1} = U^i \text{ compose } U$. A “shortcoming” in this operation is that it does not support generalization to give an approximate fixed point, if the least fixed point cannot be found.

**Generalized transitive closure.** To overcome the limitation of Omega’s transitive-closure operation, we introduced in [4] the concept of generalized transitive closure with selective generalization. Basically, it introduces generalizations of size relation based on selective grouping and hulling of its various disjuncts. While hulling aids in ensuring termination of fixed-point computation at the expense of accuracy, selective grouping and hulling help maintain accuracy of such computation.

### 3.2 Context Synthesis for Recursive Functions

For recursive functions, our analysis must derive the most informative contextual constraint that is applicable to all recursive invocations of the function, including the degenerated case. For a more accurate analysis, our method differentiates two closely-related contexts: (a) The context of a labelled call encountered during the first time the function call is invoked; i.e., before any nested recursive call is
invoked. (b) The context of a labelled call encountered after some invocations of nested recursive calls. The reason for this separation is because the latter context is computed using the synthesized recursive-call invariant.

The contextual constraint of the first call is analyzed in the same way as that for non-recursive function. For each label $L$ of a recursive function $f$, the context of the first call is:

$$ctxFst(L) = ctx(L) \land ctxSta(f)$$

where $ctx(L)$ is the derived contextual constraint at program point $L$, and $ctxSta(f)$ denotes the default context that can be assumed at procedural entry of $f$. For function $look$, $ctxSta(look) = a \geq 0$. (ie., the array must be of non-negative length.)

The contextual constraint for a labelled call encountered after subsequent recursive invocations of $f$-calls can be computed using:

$$ctxRec(L) = inverse(ctx(L)) \land inv(f) \land ctxSta(f)$$

Note that we make use of the synthesized invariant of $f$, namely $inv(f)$, while the $inverse$ operation (as defined in the Omega Calculator) is used to obtain a mirror copy of $ctx(L)$ that applies to the recursive sizes (instead of the initial sizes).

Separate identification of contexts for both the first recursive call and subsequent recursive calls is instrumental to obtaining more accurate contextual constraints, which in turns induce more precise pre-condition for eliminating recursive checks.

For the function $look$, the labels used are $L4$ and $H4$. The context enclosing the labelled call is found to be $l \leq h$. Following the above procedure, we obtain the following contextual constraints:

$$ctx(L4) = l \leq h$$
$$ctxSta(look) = a \geq 0$$
$$inverse(ctx(L4)) = l^* \leq h^*$$
$$inv(look) = a = a^* \land l \leq h, l^* \land h^* \leq h \land$$
$$2 + 2l + 2h^* \leq h + 3l^* \land l + 2h^* < h + 2l^*$$
$$ctxFst(L4) = l \leq h \land a \geq 0$$
$$ctxRec(L4) = a = a^* \land l \leq l^* \leq h^* \leq h \land 0 \leq a \land$$
$$2 + 2l + 2h^* \leq h + 3l^* \land l + 2 h^* < h + 2l^*$$

4 Pre-condition Derivation

The synthesis of contexts and invariants is essentially a forward analysis that gathers information about how values are computed and propagated and how the conditions of if-branches are inherited. In contrast, the derivation of pre-condition for check elimination is inherently a backward problem. Here, the flow of information goes from callee to caller, with the goal of finding weakest possible
pre-condition which ensures that the operation can be performed safely without checking.

We propose a backward method for deriving safety pre-conditions. This method considers each function in turn, starting from the lowest one in the calling hierarchy. Our method attempts to derive the required pre-condition to make each check redundant. Working backwards, each pre-condition that we derive from a callee would be converted into a check for its caller. In this way, we are able to derive the pre-condition for each check, including those that are nested arbitrarily deep inside procedural calls. The main steps are summarized here.

- Determine each check to see if it is either unsafe, totally redundant or partially redundant.
- Derive a safety pre-condition for each partially redundant check. Checks from recursive functions must take into account the recursive invocations.
- Amalgamate related checks together.
- To support inter-procedural propagation, convert each required pre-condition of a function into a check at its call site based on the parameter instantiation.

To help describe our method, consider the following simple example:

\[
p(arr, i, j) = \begin{cases} 
\text{if } 0 \leq i \leq j \text{ then } L6@H6@sub(arr, i) + L7@H7@sub(arr, i - 1) \\
\text{else } -1 
\end{cases}
\]

This function is a minor modification of \textit{newsub}. It takes an array and two integers \(i\) and \(j\), and returns the sum of elements at \(i\) and \(i - 1\) if \(0 \leq i \leq j\), otherwise \(-1\) is returned. From the definition of this procedure, we can provide the following sized type for \(p\):

\[
p :: (\text{Arr}^m \text{Int}, \text{Int}^i, \text{Int}^j) \rightarrow \text{Int} \\
\text{Size } m \geq 0 \land ((0 \leq i \leq j) \lor ((i < 0 \lor (i > j \land i \geq 0)) \land r = -1))
\]

### 4.1 Check Classification

We classify each check as either \textit{totally redundant}, \textit{partially redundant} or \textit{unsafe}. Given a check \(chk(L)\) under a context \(ctx(L)\), we can capture the weakest pre-condition, \(pre(L)\) that enables \(chk(L)\) to become redundant. The weakest pre-condition is computed using:

\[
pre(L) \equiv \neg ctx(L) \lor chk(L)
\]

This pre-condition should be simplified using the invariant context at procedure entry, namely \(ctxSta(p)\), whose validity would be verified by our sized-type system. If \(pre(L) \equiv True\), we classify the check as totally redundant. If \(pre(L) \equiv False\) (or unknown due to the limitation of Presburger solver), we

\[\text{In Omega, the simplification can be done by a special operator, called } gist.\]
classify the check as unsafe. Otherwise, the check is said to be partially redundant.

Example: The four checks in $p$ are:

\[
\begin{align*}
\text{chk}(L6) &= i \geq 0 & \text{chk}(H6) &= i < m \\
\text{chk}(L7) &= i - 1 \geq 0 & \text{chk}(H7) &= i - 1 < m 
\end{align*}
\]

Of these four checks, only the pre-condition of check at $L6$, namely $\text{pre}(L6) \equiv \neg \text{ctx}(L6) \lor \text{chk}(L6)$ evaluates to True. Hence, $\text{chk}(L6)$ is redundant, while the other three checks are partially redundant. In this example, we use the following contextual constraints:

\[
\begin{align*}
\text{ctx}(L6) &= \text{ctx}(L7) = \text{ctx}(H6) = \text{ctx}(H7) \\
\text{ctx}(L6) &= \text{ctxSta}(p) \land (0 \leq i \leq j) \quad \text{and} \quad \text{ctxSta}(p) = m \geq 0
\end{align*}
\]

4.2 Derivation of Pre-condition

The derivation of $\text{pre}(L)$ is to a large extent dependent on $\text{ctx}(L)$. A more informative $\text{ctx}(L)$ could lead to a better $\text{pre}(L)$. For a given contextual constraint $\text{ctx}(L)$, $\text{pre}(L)$ can be computed by:

\[
\text{pre}(L) \equiv \neg \text{ctx}(L) \lor \text{chk}(L)
\]

The following lemma characterizes $\text{pre}(L)$ as the weakest pre-condition. We omit the proof in this paper.

**Lemma 1** The weakest pre-condition ($\text{pre}$) for the safe elimination of a check ($\text{chk}$) in a given context ($\text{ctx}$) is $\text{pre} \equiv \neg \text{ctx} \lor \text{chk}$.

Example: Using the above formulae, we can derive the following pre-conditions for the three partially redundant checks:

\[
\begin{align*}
\text{pre}(H6) &= (i \leq -1) \lor (j < i \land 0 \leq i) \lor (i < m) \\
\text{pre}(L7) &= (i \leq -1) \lor (j < i \land 0 \leq i) \lor (i \geq 1) \\
\text{pre}(H7) &= (i \leq -1) \lor (j < i \land 0 \leq i) \lor (i \leq m)
\end{align*}
\]

Deriving pre-conditions for the elimination of checks from recursive procedure is more challenging. A key problem is that the check may be executed repeatedly, and any derived pre-condition must ensure that the check is completely eliminated. One well-known technique for the elimination of checks from loop-based program is the loop limit substitution method of [7]. Depending on the direction of monotonicity, the check of either the first or last iteration of the loop is used as a condition for the elimination of all checks. However, this method is restricted to checks on monotonic parameters whose limits can be precisely calculated.

We propose a more general method to handle recursive checks. For better precision, our approach separates out the context of the initial recursive call from the context of the subsequent recursive calls. The latter context may use the invariant of recursive parameters from sized typing.
Using the recursive look function (whose parameters are non-monotonic) as an example, we shall provide two separate checks for the first and subsequent recursive calls, namely:

\[ \text{chkFst}(L4) = 0 \leq l + h \quad \text{and} \quad \text{chkRec}(L4) = 0 \leq l^* + h^* \]

with their respective contexts:

- \( \text{ctxFst}(L4) = a \geq 0 \land l \leq h \)
- \( \text{ctxRec}(L4) = a \geq 0 \land l^* \leq h^* \land \text{inv}(\text{look}) \)
- \( \text{inv}(\text{look}) = a = a^* \land l \leq l^*, h \land h^* \leq h \land 2 + 2l + 2h^* \leq h + 3l^* \land l + 2h^* < h + 2l^* \)

We next derive the pre-conditions for the two checks separately, as follows:

- \( \text{preFst}(L4) = \neg \text{ctxFst}(L4) \lor \text{chkFst}(L4) = (h < l) \lor (0 \leq l + h) \)
- \( \text{preRec}(L4) = \neg \text{ctxRec}(L4) \lor \text{chkRec}(L4) = \forall l^*, h^*. (a \geq 0 \land l^* \leq h^* \land l \leq h, l^* \land h^* \leq h \land 2 + 2l + 2h^* \leq h + 3l^* \land l + 2h^* < h + 2l^*) \lor (0 \leq l^* + h^*) \)

We can now combine the two pre-conditions together in order to obtain a single safety pre-condition for the recursive check, as shown here:

\[ \text{pre}(L4) = \text{preFst}(L4) \land \text{preRec}(L4) = (h < l) \lor (0 \leq l + h \land 0 \leq l) \]

Through a similar derivation, the other check of \( H4 \), based on the precondition \( l + h < 2a \) from \( \text{getMid} \), yields:

\[ \text{pre}(H4) = \text{preFst}(H4) \land \text{preRec}(H4) = (h < l) \lor (h < a \land l + h < 2a) \]

The derived pre-conditions are very precise. Apart from ensuring that the given recursive checks are safe, it also captures a condition on how the checks may be avoided.

### 4.3 Amalgamating Related Checks

As some of the checks are closely related, it may be useful to amalgamate these checks together. At the risk of missing out some opportunities for optimization, the amalgamation of related checks serves two purposes, namely:

\(^3\) In general, any two checks can be amalgamated together. However, closely related checks will have a higher probability of being satisfied at the same time. This can help ensure amalgamation without loss of optimization.
- It can cut down the time taken for our analysis.
- It can reduce the number of specialization points, and hence the size of the specialized code.

We propose a simple technique to identify related checks. Given two checks \( C_1 \) and \( C_2 \), we consider them to be related if either \( C_1 \Rightarrow C_2 \) or \( C_2 \Rightarrow C_1 \). For example, checks \( H_6 \) and \( H_7 \) are related since \( \text{chk}(H_6) \Rightarrow \text{chk}(H_7) \). Because of this similarity, we can combine the pre-conditions of these two checks, as follows:

\[
\text{pre}(H_6, H_7) = \text{pre}(H_6) \land \text{pre}(H_7) = i \leq -1 \lor (j < i \land 0 \leq i) \lor i < m
\]

The combined pre-condition can eliminate both checks simultaneously.

### 4.4 Inter-procedural Propagation of Checks

To support inter-procedural propagation of checks, each pre-condition for a partially redundant check must first be converted into a new check at the call site. After that, the process of classifying the check and deriving its safety pre-condition is repeated.

Consider two functions:

\[
f(v_1, ..., v_n) = \ldots L \circ \text{sub}(\text{arr}, i) \ldots
g(w_1, ..., w_n) = \ldots C \circ f(v_1', ..., v_n') \ldots
\]

Suppose that in \( f \), we have managed to derive a non-trivial \( \text{pre}(L) \) that would make \( \text{chk}(L) \) redundant. Then, at each call site of \( f \), such as \( f(v_1', ..., v_n') \) in the body of function \( g \), we should convert the pre-condition of \( f \) into a new check at the call site, as follows:

\[
\begin{align*}
\text{chk}(C) &= \exists X. \text{pre}(L) \land \text{subs}(C) \\
\text{subs}(C) &= \land_{i=1}^{n} (\text{eq } \tau_i \tau_i') \quad \text{where } v_i :: \tau_i; v_i' :: \tau_i'; X = \cup_{i=1}^{n} f v(\tau_i)
\end{align*}
\]

The pre-condition of \( f \) is converted into a check via a size parameter substitution, \( \text{subs}(C) \).

Example: Consider a function \( q \):

\[
q :: (\text{Arr}^n \text{Int}, \text{Int}^k) \rightarrow \text{Int}^s
\]

\[
q(\text{arr}, k) = \text{let } r = \text{random}(); l = k + 1 \text{ in } C8 \circ C9 \circ p(\text{arr}, r, l)
\]

At the labelled call site, we have:

\[
\text{subs}(C8) = \text{subs}(C9) \quad \text{and} \quad \text{subs}(C9) = (m = n) \land (j = l) \land (i = r)
\]

We assume that the size variables assigned to the arguments of the \( p \) call are \( n, r \) and \( l \), respectively. Using our formula for converting the pre-condition of \( p \) into a check at its call site, we obtain:
\begin{align*}
\text{chk}(C8) &= \exists i, j. \text{pre}(L7) \land \text{subs}(C8) \\
&= (r \leq -1) \lor (l < r \land 0 \leq r) \lor (r \geq 1) \\
\text{chk}(C9) &= \exists i, j. \text{pre}(H6, H7) \land \text{subs}(C9) \\
&= (r \leq -1) \lor (l < r \land 0 \leq r) \lor (r < n)
\end{align*}

With this, we can propagate the check backwards across the procedure of \textit{q} by deriving the following two pre-conditions.

\begin{align*}
\text{pre}(C8) &= \forall r, l. \neg (l = k + 1) \lor ((r \leq -1) \lor (l < r \land 0 \leq r) \lor (r \geq 1)) \\
&= k \leq -2 \\
\text{pre}(C9) &= \forall r, l. \neg (l = k + 1) \lor ((r \leq -1) \lor (l < r \land 0 \leq r) \lor (r < n)) \\
&= (k \leq -2) \lor (-1 \leq k \leq n - 2)
\end{align*}

Note that since \( r \) and \( l \) are local variables; we must eliminate them from our pre-condition by using universal quantification. Universal quantification ensures that we get a new pre-condition that is safe for all values of \( r \) and \( l \).

Inter-procedural propagation of checks applies to recursive functions without exception.

Example : The pre-condition for \textit{look} can be converted to checks at its call site in \textit{bsearch}, as follows:

\begin{align*}
\text{chk}(L5) &= \exists l, h. \text{pre}(L4) \land \text{subs}(L5) \\
&= \exists l, h. ((h < l) \lor (0 \leq l + h \land 0 \leq l)) \land (l = 0 \land h = v - 1) \\
&= (v \leq 0) \lor (1 \leq v) \\
\text{chk}(H5) &= \exists l, h. \text{pre}(H4) \land \text{subs}(L5) \\
&= \exists l, h. ((h < l) \lor (h < a \land l + h < 2a)) \land (l = 0 \land h = v - 1) \\
&= (v \leq 0) \lor (v \leq a, 2a) \\
\text{subs}(L5) &= (l = 0) \land (h = v - 1)
\end{align*}

From here, we can derive the safety pre-conditions for \textit{bsearch} as shown below.

\begin{align*}
\text{pre}(L5) &= \neg \text{ctx}(L5) \lor \text{chk}(L5) \\
&= \forall v. \neg (v = a \land a \geq 0) \lor (v \leq 0 \lor 1 \leq v) \\
&= \text{True} \\
\text{pre}(H5) &= \neg \text{ctx}(H5) \lor \text{chk}(H5) \\
&= \forall v. \neg (v = a \land a \geq 0) \lor (v \leq 0 \lor v \leq a, 2a) \\
&= \text{True}
\end{align*}

Through this inter-procedural propagation, we have successfully determined that the recursive checks of \textit{look} inside \textit{bsearch} are totally redundant. Hence, all bound checks for \textit{bsearch} can be completely eliminated. This is done by providing specialized versions of \textit{look} and \textit{getmid} (without bound checks) that would be called from \textit{bsearch}. 
5 Bound Check Specialization

With the derived pre-condition for each partially redundant check, we can now proceed to eliminate more bound checks by specializing each call site with respect to its context. The apparatus for bound check specialization is essentially the same as contextual specialization [6, 16] where each function call can be specialized with respect to its context of use. A novelty of our method is the use of derived pre-condition to guide specialization. This approach is fully automatic and can give better reuse of specialized code.

Suppose that we have a function \( f \) with \( N \) checks, that is used in one of its parent function \( g \) as follows:

\[
f(v_1, \ldots, v_n) = t_f \mathbf{Req} \{ P_i \}_{i=1}^{N}
g(v_1, \ldots, v_n) = \ldots \{ C_i \}_{i=1}^{N} @ f(v'_1, \ldots, v'_n) \ldots
\]

Notation-wise, we write \( \{ C_i \}_{i=1}^{N} @ f(v'_1, \ldots, v'_n) \) as the short form for \( C_1 @ \ldots @ C_N @ f(v'_1, \ldots, v'_n) \).

Suppose further that we have a context \( ctx(C) \), for the labelled call, which may encompass a context that could be inherited from the specialization of \( g \). Let the set of pre-conditions whose checks could be made redundant be:

\[
G = \{ P_i | i \in 1 \ldots N \land ctx(C) \Rightarrow \text{chk}(C_i) \}
\]

For maximal bound check optimization, we should specialize each of the call for \( f \) to a version that would maximize bound check elimination. In the above example, our specialization would introduce \( f_G \), as follows:

\[
g(v_1, \ldots, v_n) = \ldots f_G(v'_1, \ldots, v'_n) \ldots
f_G(v_1, \ldots, v_n) = S[t_f] G \quad \text{where ctxSta}(f_G) = G \land ctxSta(f)
\]

Note how the context \( G \), which contains the maximum pre-conditions that are satisfiable in \( ctx(C) \), is propagated inside the body of \( f \) by specializer \( S \). This specialization is commonly known as polyvariant specialization. It will generate a specialized version of the code for each unique set of checks that can be eliminated. It can provide as many variants of the specialized codes as there are distinguishable contexts. To minimize the number of variants used, the specialization process will proceed top-down from the main function, and generate a specialized version only if it is required directly (or indirectly) by the main function. Polyvariant specialization can help maximize the elimination of bound checks. However, there is a potential explosion of code size, as the maximum number of specialized variants for each function is \( 2^N \) where \( N \) is the number of partially redundant checks that exist. In practice, such code explosion seldom occur, unless the function is heavily reused under different contexts.

If code size is a major issue (say for embedded systems), we could use either monovariant specialization or duovariant specialization.

In monovariant specialization, we will need an analysis technique to help identify the best common context, call it \( ctxMin(f) \), that is satisfied by all the call sites. Let the set of call sites to \( f \) in a given program be:
\{\{C_{ij}\}_{i=1}^N @ f(v_{j1}, ..., v_{jn})\}_{j=1}^M.

and their corresponding contexts be \{ctx(C_j)\}_{j=1}^M. We define the best common context of these call sites to be:

\[ctxMin(f) = \{P_i | i \in 1..N, \forall j \in 1..M. ctx(C_j) \Rightarrow chk(C_{ij})\}\]

With this most informative common context, we could now provide a least specialized variant for \(f\) that could be used by all call sites in our program, as follows:

\[
f_{min}(v_1, ..., v_n) = S[t_f] \text{ctxMin}(f) \quad \text{where}
ctxSta(f_{min}) = \text{ctxMin}(f) \land ctxSta(f)
\]

For duovariant specialization, we shall generate a version of each function \(f\) that is maximally specialized, namely:

\[
f_{max}(v_1, ..., v_n) = S[t_f] \text{ctxMax}(f) \quad \text{where}
ctxSta(f_{max}) = \text{ctxMax}(f) \land ctxSta(f)
ctxMax(f) = \{P_i | i \in 1..N, \exists j \in 1..M. ctx(C_j) \Rightarrow chk(C_{ij})\}
\]

This most specialized variant should be used whenever possible. With the three variants of bound check specialization, we now have a spread of the classic space-time tradeoff. We hope to investigate the cost-effectiveness of these alternatives in the near future.

6 Performance Analysis

In this section, we address the practicality of using constraint solving for implementing both our forward analysis (essentially sized typing) and backward analysis (for deriving pre-conditions).

Our experiments were performed with Omega Library 1.10, running on a Sun System 450. We took our examples mostly from [26], with the exception of sumarray from [27]. The reported time measurements are the average values out of 50 runs. The first column reports the time taken by forward analysis (largely for computing invariants), while the second column reports the time taken for backward derivation of safety pre-condition.

The results shows that the time taken by the analyses required by array bound checks optimization are largely acceptable. A slightly higher analysis time was reported for hanoi, due largely to the more complex recursive invariant being synthesized.

Our analysis determines that all checks in these examples are totally redundant. Consequently, they are eliminated in the specialized codes. Gains in run-time efficiency range between 8% (for “sumarray” program) and 56% (“matrix mult” program), which is comparable to those found in the literature (such as [26]).
7 Related Work

Traditionally, data flow analysis techniques have been employed to gather available information for the purpose of identifying redundant checks, and anticipatable information for the purpose of hoisting partially redundant checks to a more profitable location. The techniques employed have gradually increased in sophistication, from the use of family of checks in [15], to the use of difference constraints in [3]. While the efficiency of the techniques are not in question, data flow analysis techniques are inadequate for handling checks from recursive procedures, as deeper invariants are often required.

To handle checks from programs with more complex control flow, verification-based methods have also been advocated by Suzuki and Ishihata [24], Necula and Lee [19, 20] and Xu et al [27]; whilst Cousot and Halbwachs [5] have advocated the use of abstract interpretation techniques. Whilst powerful, these methods have so far been restricted to eliminating totally redundant checks.

It is interesting to note that the basic idea behind the backward derivation of weakest pre-condition was already present in the inductive iteration method, pioneered by Suzuki and Ishihata [24], and more recently improved by Xu et al [27]. However, the primary focus has been on finding totally redundant checks. Due to this focus, the backward analysis technique proposed in [24] actually gathers both pre-condition and contextual constraints together. Apart from missing out on partially redundant checks, their approach is less accurate than forward methods (such as [5]) since information on local variables are often lost in backward analysis.

Xi and Pfenning have advocated the use of dependent types for array bound check elimination [26]. While it is possible to specify pre-conditions through dependent types, they do not specially handle partially redundant checks. Moreover, the onus for supplying suitable dependent types rest squarely on the programmers.

Recently, Rugina and Rinard [23] proposed an analysis method to synthesize symbolic bounds for recursive functions. In their method, every variable is expressed in terms of a lower and an upper symbolic bound. By assuming a polynomial form for the symbolic bounds, their method is able to compute these
bounds without using fix-point iteration. In some sense, this inductive approach is similar to the proposal made in \[25, 21\], where size information is inductively captured to analyze program termination property. Whilst the efficiency of the inductive approach is not in question, we have yet to investigate the loss in precision that come with fixing the expected target form.

8 Conclusion and Future Work

Through a novel combination of both forward technique to compute contextual constraint, and backward method to derive weakest pre-conditions, we now have a comprehensive method for handling both totally redundant and partially redundant checks. Both analysis methods are built on top of a Presburger constraint solver that has been shown to be both accurate and practically efficient \[22\]. Our new approach is noteworthy in its superior handling of partially redundant checks.

There are several promising directions for future research. They deal largely with how the precision of optimization and efficiency of analysis method could be further improved.

Firstly, our contextual constraint analysis presently inherits its constraints largely from conditional branches. We can further improve its accuracy by propagating prior bounds checks in accordance with the flow of control. For this to work properly, we must be able to identify the weakest pre-conditions for each function that could be asserted as post-condition, after each call has been successfully executed. As bound errors could be caught by exception handling, the extent of check propagation would be limited to the scope where the bound errors are uncaught.

Secondly, not all partially redundant checks could be eliminated by its caller’s context. Under this scenario, it may be profitable to insert speculative tests that could capitalize on the possibility that safety pre-condition are present at runtime. Whilst the idea of inserting speculative runtime test is simple to implement, two important issues that need to be investigated are (i) what test to insert, and (ii) where and when will it be profitable to insert the selected test. Specifically, we may strip out the avoidance condition from the speculative test, and restricts such runtime tests\textsuperscript{4} to only recursive checks.

Lastly, the efficiency of our method should be carefully investigated. The cost-benefit tradeoff of check amalgamation and bound check specialization would need to be carefully studied in order to come up with a practically useful strategy. Also, the sophistication (and cost) of our approach is affected by the type of constraints that is supported. Whilst Presburger formulae have been found to be both precise and efficient, it may still be useful to explore other types of constraint domains.

\textsuperscript{4} The insertion of speculative tests may look similar to check hoisting. The key different is that no exception is raised if speculative test fails.
9 Acknowledgment

We would like to thank the anonymous referees for their valuable comments. This work has been supported by the research grants RP3982693 and RP3991623.

References

Type Systems for Useless-Variable Elimination

Adam Fischbach and John Hannan

Department of Computer Science and Engineering
The Pennsylvania State University, University Park, PA 16802, USA
{fischbac, hannan}@cse.psu.edu

Abstract. We present a type-based specification for useless-variable elimination for a higher-order, call-by-value functional language. Utilizing a weak form of dependent types, we introduce a mechanism for eliminating at runtime useless code that is not detectable statically. We prove the specifications sound and safe with respect to an operational semantics, ensuring that eliminated expressions contributed no observable behavior to the original program. We define an algorithm that implements useless-variable elimination without dependent types, and we prove this algorithm correct with respect to the specification.

1 Introduction

A variable can be considered useless if its value contributes nothing to the behavior of the program. Code can be considered useless if its computation and value (for code corresponding to expressions) contribute nothing to the behavior of a program. Useless variables and code arise from various program transformations (e.g., arity raising, partial evaluation), from the maintenance and evolution of software, and from programs extracted from proofs. Useless-variable elimination (UVE) and useless-code elimination (UCE) are the operations of removing useless formal parameters (variables) and corresponding actual parameters (code) from a program. (More generally, useless code elimination can also include detecting and eliminating dead or unreachable code.) These optimizations reduce the number of arguments passed to a function, eliminate unneeded computations, and potentially shorten the lifetime of some variables. All of these aspects contribute to the importance of UVE as an optimization of program representation.

We study UVE and UCE using an approach based on type inference in which types convey information about the use or need of variables and code. We demonstrate that by considering richer notions of types we can detect and eliminate more occurrences of useless items. We begin with a type system based on simple types with a very simple form of subtyping. This specification can detect and eliminate many useless variables and expressions, but the type system also imposes too many constraints, prohibiting the identification of additional occurrences of useless items. We then extend the types and type system to include...
a weak form of dependent types. Not only does this allow us to identify more useless variables and expressions, it also allows us to identify expressions that can be ignored at run time even though they cannot be eliminated statically. Our use of dependent types is closely related to the use of conjunctive types to perform UCE \cite{2}.

Unlike flow-based approaches \cite{8,9}, type-based ones are relatively straightforward and reasonably efficient. Except for the restriction to a typed language, our approach (and other typed-based ones) can detect and eliminate more useless code than existing flow-based approaches. The correctness proofs for our approach (see \cite{4}) are significantly simpler than the proofs for flow-based approaches.

Unlike other type-based approaches to this problem \cite{7,3}, we apply annotations to function arrows to indicate the use/need of a function’s parameter and corresponding arguments. These other approaches consider replacing the type of a useless variable or expression with a type that indicates uselessness. Consider the function \( \lambda x : \text{int.} \). While other approaches use type inference to compute a type such as \((\text{unit} \rightarrow \text{int})\) \cite{7} and \((\omega \rightarrow \delta \rightarrow \text{int})\) \cite{3}, we infer a type \((\text{int} \rightarrow u \text{ int})\). For many kinds of examples, these approaches are roughly equivalent in their expressive power, but as we include richer type features, like dependent types, our approach appears to offer a simpler and more powerful framework for detecting more useless variables and code.

The remainder of the paper is organized as follows. In the next section we introduce a simple, typed functional language that we use for both the source and target of our translation. The language contains annotations regarding useless variables and expressions. In Section 3 we define useless-variable (and useless-code) elimination via a type system that enforces constraints regarding use between types and terms. We demonstrate the correctness of this specification with respect to the operational semantics. In Section 4 we consider an extension to our simple-type specification that includes dependent types. We motivate the need for improvements to our original specification and justify how this extension increases the ability to detect more useless variables and code. In Section 5 we present and prove correct an algorithm for UVE without dependent types. We conclude in Section 6.

## 2 Language Definition

We introduce a simple, typed, higher-order functional language with recursive function definitions. To support the specification of useless code and useless variables, the language assumes a set of dummy variables, indexed by type, and two forms of application.

\[
e :: = n \mid d^\tau \mid x \mid \lambda x.e \mid e @ n e \mid e @ u e \mid \text{if} z e e e \mid \mu f.\lambda x.e \mid e + e
\]

Each dummy variable is annotated with its simple type \( \tau \). The annotations \( n \) and \( u \) indicate possibly needed and definitely unneeded operands, respectively.
The goal of UVE is to replace occurrences of `@n` with `@u` where it is safe to do so.

We assume a call-by-value operational semantics for this language axiomatized by a judgment of the form $\rho \triangleright e \leftrightarrow v$, in which $\rho$ is an environment, $e$ is an expression and $v$ is a value. The rules are almost entirely standard except for application and recursion. For application, we have two rules corresponding to the two forms of annotations:

$$
\frac{
\rho \triangleright e_1 \leftarrow [\rho', \lambda x.e] \quad \rho \triangleright e_2 \leftarrow v_2 \quad \rho\{x \mapsto v_2\} \triangleright e \leftrightarrow v
}{\rho \triangleright (e_1 \at n e_2) \leftrightarrow v}
$$

$$
\frac{
\rho \triangleright e_1 \leftarrow [\rho', \lambda x.e] \quad \rho \triangleright e \leftrightarrow v
}{\rho \triangleright (e_1 \at u e_2) \leftrightarrow v}
$$

For handling recursion, we use substitution instead of environments:

$$
\frac{
\rho \triangleright \mu f.\lambda x.e \leftrightarrow [\rho, \lambda x.e[\mu f.\lambda x.e/f]]
}{\rho \triangleright e_1 \leftarrow [\rho', \lambda x.e] \quad \rho \triangleright e \leftrightarrow v}
$$

This formulation of operational semantics provides a convenient setting in which to reason about the correctness of our work.

We can restrict our attention to a sublanguage of this language that does not contain dummy variables or applications annotated by $u$. The resulting language is a traditional, typed functional language. We drop the $n$ annotation from applications since it is superfluous in this sublanguage. We consider this sublanguage as the input language to our UVE analysis.

The type system for our language is almost entirely standard, axiomatized by a judgment of the form $\Gamma \triangleright e : \tau$. The types consist of simple types. The annotations on applications are ignored by the type system. We include a rule for typing dummy variables:

$$
\frac{
}{\Gamma \triangleright d : \tau}
$$

## 3 UVE with Simple Types

We give a formal specification of the UVE transformation using simple types annotated with use information. We axiomatize a deductive system that relates a source term, its annotated type, and a target term (a UVE-form of the source term). The inference rules follow the structure of a traditional type system for simple types with additional constraints supporting UVE.

### 3.1 UVE Types

As in previous work [5, 6], we use annotations on types to provide a more detailed characterization of expressions: the type of an expression will convey information regarding the use of variables in the expression. The inclusion of these annotations on inference rules in a type system introduces constraints which ensures that these need properties are valid. (This validity is provided by a type
For UVE, we use annotations corresponding to *needed* and *unneeded* (useless) variables:

\[ a ::= n \mid u \]

The annotation \( a \) on a function type indicates the approximate need of the formal parameter of a function of that type.

\[ \tau ::= \text{int} \mid \tau \to a \tau \]

A function of type \( \tau_1 \to u \tau_2 \) does not need its argument, e.g., \( \lambda x.1 : \tau \to u \text{int} \), and a function of type \( \tau_1 \to n \tau_2 \) may need its argument, e.g., \( \lambda x.x+1 : \text{int} \to n \text{int} \).

As should be expected, we must conservatively estimate the need of a variable, possibly indicating that a variable is needed when, in fact, it may not be. To increase the precision of this approximation we define an ordering on types.

**Definition 1.** Let \( a \leq a' \) and \( \tau \leq \tau' \) be defined by the following rules:

\[
\begin{align*}
    u &\leq n & a &\leq a & \tau_1 \leq \tau_1' & \tau_2 \leq \tau_2' & a \leq a' \\
    \text{int} &\leq \text{int} & (\tau_1 \to a \tau_2) &\leq (\tau_1' \to a' \tau_2')
\end{align*}
\]

Intuitively, if \( \tau \leq \tau' \) and we have an expression \( e \) of type \( \tau \), then we can also consider (use) \( e \) as an expression of type \( \tau' \).

### 3.2 Specification of UVE

With these types defined we can introduce useless-variable elimination as a type-based deductive system. We introduce the judgment \( \Gamma \vdash e : \tau \Rightarrow e' \) in which \( \Gamma \) is a type context, \( e \) is an expression in our input language, \( e' \) is an expression with all detected useless code eliminated, and \( \tau \) is the annotated type of \( e' \). The judgment is axiomatized by the rules in Figure 1. We assume the expression \( e \) is well-typed (in the language’s type system).

We have two rules each for \( \lambda \)-abstractions and applications: one for the elimination case (eliminating a formal parameter or argument) and one for the default case (no elimination). In the first rule for abstraction (eliminating a parameter) we use the condition that \( x \) not occur free in the translation of the function’s body to determine that the parameter is useless. The second rule for abstractions is straightforward.

The two rules for handling application use type information to ensure that the annotation on applications is consistent with the type of the operator. A type consistency result will ensure that if \( e_1 \) has type \( \tau_1 \to u \tau_2 \) then its value will be a function of that type, i.e., a function that does not need its argument. The operand of an application annotated by \( u \) must have no effect, otherwise eliminating it might change the behavior of the program. In our language, the only expressions that generate effects are addition (due to the possibility of overflow) and recursion (due to the possibility of nontermination). We assume a precomputed effect analysis on programs. We refrain from presenting it here.
Type Systems for Useless-Variable Elimination 29

\[ \Gamma \vdash n : \text{int} \Rightarrow n \quad \Gamma \vdash x : \tau \Rightarrow x \]

\[ \Gamma \vdash \lambda x.e : \tau_1 \rightarrow u \tau_2 \Rightarrow \lambda x.e' \quad \Gamma \vdash \lambda x.e : \tau_1 \rightarrow n \tau_2 \Rightarrow \lambda x.e' \]

\[ \Gamma \vdash \mu f.\lambda x.e : \tau \Rightarrow \mu f.\lambda x.e' \]

\[ \Gamma \vdash e_1 : \text{int} \Rightarrow e'_1 \quad \Gamma \vdash e_2 : \tau_2 \Rightarrow e'_2 \quad \tau_2 \leq \tau_2 \]

\[ \Gamma \vdash (e_1 @ e_2) : \tau \Rightarrow (e'_1 @ n e'_2) \]

\[ \Gamma \vdash (e_1 + e_2) : \text{int} \Rightarrow (e'_1 + e'_2) \]

or including it as part of the UVE analysis, as this approach only serves to complicate the presentation of useless-variable elimination while contributing very little to the understanding of useless variables.

As an example of the capabilities of our specification consider the following term in which \( N \) and \( M \) are some effect-free terms. (The use of let is only for improved clarity.)

```plaintext
let f1 = \lambda x.3
f2 = \lambda x.x+1
g = \lambda k. (k @ N)
h = \lambda k. (k @ N)
in (g @ f1) + (g @ f2) + (f1 @ M) + (h @ f1) end
```

Our specification produces the following expression:

```plaintext
let f1 = \lambda x.3
f2 = \lambda x.x+1
g = \lambda k. (k @ n N)
h = \lambda k. (k @ u d\text{int})
in (g @ n f1) + (g @ n f2) + (f1 @ u d\text{int}) + (h @ n f1) end
```

Even though \( f1 \) and \( f2 \) are both passed as arguments to \( g \), they do not need to have identical types. They only need to have a common subtype. But the application \( (k @ N) \) in the body of \( g \) needs its argument because \( f2 \) needs its argument. The application \( (k @ N) \) in the body of \( h \) does not need its argument because \( f1 \) does not need its argument.
3.3 Correctness

We consider two aspects of correctness. First, because we consider a typed language, we ensure that the results of the UVE transformation are always well-typed programs. This result trivially follows from the structure of our specification because it enforces the traditional simple-type rules and the transformation only replaces variables and expressions with other variables and expressions of the same type. Second, we ensure that the transformation preserves the behavior of programs. Because we represent both the language’s definition (type system and operational semantics) and the UVE analysis/transformation as deductive systems, relating these is straightforward and requires few auxiliary results.

For the operational correctness of our analysis and transformation, we want to ensure that a source program and its transformed (UVE-eliminated) form have the same behavior. This means that one must evaluate to a value iff the other also evaluates to that value (at a base type). Our theorem incorporates such an equivalence between terms and also a type consistency result. The latter is actually crucial to proving the former.

To express a statement of operational equivalence we need to define a relation between values and a relation among environments and contexts:

Definition 2. The relations \( v : \tau \Rightarrow v' \) and \( \rho : \Gamma \Rightarrow \rho' \) are the least relations satisfying the following conditions:

1. \( n : \text{int} \Rightarrow n \) (for all integers \( n \));
2. \( [\rho, \lambda x.e] : \tau \Rightarrow [\rho', e'] \) if there exists \( \tau' \) and \( \Gamma \) such that \( \rho : \Gamma \Rightarrow \rho' \), \( \tau' \leq \tau \), and \( \Gamma \vdash \lambda x.e : \tau' \Rightarrow e' \) is derivable;
3. \( \rho : \Gamma \Rightarrow \rho' \) if \( \text{dom}(\rho) = \text{dom}(\Gamma) \), \( \text{dom}(\rho') \subseteq \text{dom}(\rho) \) and for all \( x \in \text{dom}(\rho') \), \( \rho(x) : \Gamma(x) \Rightarrow \rho'(x) \).

In the second case we use \( e' \) for the term in the closure corresponding to \( v' \) in anticipation of developments in the next section. For the purposes of the current section, we could have used \( \lambda x.e' \) instead. The justification for the condition \( \tau' \leq \tau \) in the second case is that we might have an expression like \( (k \mathbin{@\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!n) \) \) in which \( k \) has type \( \text{int} \rightarrow \text{int} \) but is bound to a value \( (f1) \) of type \( \text{int} \rightarrow_{\text{u}} \text{int} \) (because, for example, the application rule allows a function \( \lambda x.e \) of type \( \tau_1 \rightarrow_{\text{u}} \tau_2 \) to be applied to an argument \( e' \) of some type \( \tau'_1 \) such that \( \tau'_1 \leq \tau_1 \)). Note in the last case we only have \( \text{dom}(\rho') \subseteq \text{dom}(\rho) \). This inequality reflects the eliminated useless variables in the translated term.

Theorem 1. If \( \Gamma \vdash e : \tau \Rightarrow m \), \( \rho : \Gamma \Rightarrow \rho' \), and \( \text{FV}(m) \subseteq \text{dom}(\rho') \) then

1. if \( \rho \vdash e \leftarrow v \) then there exists a \( v' \) such that \( \rho' \vdash m \leftarrow v' \) and \( v : \tau \Rightarrow v' \);
2. if \( \rho' \vdash m \leftarrow v' \) then there exists a \( v \) such that \( \rho \vdash e \leftarrow v \) and \( v : \tau \Rightarrow v' \).

Part 1 expresses the soundness of our analysis and part 2 expresses its safety. The proof of both parts can be found in [4].

As a special case of the theorem, we consider closed terms at base type:

Corollary 1. If \( \cdot \vdash e : \text{int} \Rightarrow m \) then \( \cdot \vdash e \leftarrow n \) iff \( \cdot \vdash m \leftarrow n \).
4 UVE with Dependent Function Types

While simple types with subtyping detects and eliminates many instances of useless variables and code, the typing constraints imposed by that system can still force a function to have some type $\tau$ due to its context, but when considered by itself it has a type strictly less than $\tau$.

4.1 Motivating the Need for Dependent Types

Consider the following example [1]:

```
let f = \h. \lambda z.(h \ @ z)
in (f \ @ (\lambda x.3) \ @ N) + (f \ @ (\lambda x.x) \ @ 7)
```

in which function $f$ has type $(\text{int} \to \text{int}) \to \text{int} \to \text{int}$, and variables $x$, $y$, $z$, and $h$ have type $\text{int}$, $\text{int}$, $\text{int}$, and $(\text{int} \to \text{int})$, respectively.

Because $f$ is bound to a term that applies its first argument to its second, the term $N$ is useless code. (We again assume $N$ to be an effect-free term.) To identify this, the first occurrence of $f$ should have type $(\text{int} \to_u \text{int}) \to_\eta \text{int} \to_u \text{int}$, while the second occurrence of $f$ must have type $(\text{int} \to_\eta \text{int}) \to_\eta \text{int} \to_\eta \text{int}$. Each of these is a type that can be assigned to the term $(\lambda h. \lambda z.(h \ @ z))$ when considered by itself. But in the context of the example above, the term can only be assigned the latter type since $(\lambda x.x)$ has type $\text{int} \to_\eta \text{int}$, which is not a subtype of $\text{int} \to_u \text{int}$. So in the system described in Figure 1 we would be forced to identify $N$ as needed.

The two types of $f$ given above are, however, both instances of a more general type involving *annotation variables*: $(\text{int} \to_\gamma \text{int}) \to_\eta \text{int} \to_\gamma \text{int}$. This observation might lead one to pursue a notion of polymorphism for annotations, leading to a type such as $\forall \gamma. (\text{int} \to_\gamma \text{int}) \to_\eta \text{int} \to_\gamma \text{int}$. However, the translated body of $f$ would be $(h \ @_\gamma z)$ and the type system would not allow us to assign a meaning to the annotation. We observe that the need of the operand depends on the input to the function. This leads us to consider a form of dependent types involving the quantification of annotation variables that allows us to explicitly pass annotations to functions. Alternatively, we could view this approach as a form of explicit polymorphism in which the quantification is only over annotation variables (not type variables).

This approach is in contrast to the use of conjunctive types by Damiani [2], though the ideas are similar. In [7], Kobayashi uses ML-style let polymorphism to handle the above example since he does not make use of annotations but rather assigns the type $\text{unit}$ to useless expressions.

To handle dependent types we need to extend the definition of annotations, types, terms, and operational semantics. We also distinguish between two kinds of useless code: static and dynamic. Static useless code is what we have previously identified as useless code: code that can statically be determined to be useless. Dynamic useless code is code that can be determined useless at runtime and therefore an implementation can avoid executing it. Some dynamically useless
code might always be useless or it might sometimes be useful. In either case, we cannot statically determine that it is always useless and so we cannot eliminate it from the translated term.

### 4.2 A Language with Dependent Types

We extend the definition of annotations and types to include annotation variables (ranged over by $\gamma$) and dependent types that abstract annotation variables:

\[
\begin{align*}
a &::= u \mid n \mid \gamma \\
\tau &::= \text{int} \mid \tau \rightarrow_a \tau \mid \Pi \gamma.\tau
\end{align*}
\]

We also extend the definition of the language to support annotation-variable abstraction and application:

\[
e ::= \cdots \mid \lambda \gamma. e \mid e \at a
\]

Along with this extension of the syntax we give new inference rules for the static and dynamic semantics of the new features:

\[
\begin{align*}
\Gamma \vdash e : \tau &\quad \gamma \not\in \text{FV}(\Gamma) \\
\Gamma \vdash \lambda \gamma.e : \Pi \gamma.\tau \\
\Gamma \vdash e : \Pi \gamma.\tau \\
\Gamma \vdash e @ a : \tau[a/\gamma]
\end{align*}
\]

\[
\begin{align*}
\rho \vdash e \leftrightarrow [\rho', \lambda \gamma.e'] &\quad \rho' \vdash e'[a/\gamma] \leftrightarrow v \\
\rho \vdash (e @ a) \leftrightarrow v
\end{align*}
\]

We use substitution, instead of bindings in an environment, for handling mapping annotation variables to values to maintain a simple environment structure.

Using this extended language we can modify the example above to incorporate explicit annotation abstraction and application:

\[
\begin{align*}
\text{let } f & = \lambda \gamma. \lambda h. \lambda z.(h @ \gamma z) \\
\text{in } (f @ u @ n (\lambda x. 3) @ u \text{ int}) + (f @ n @ n (\lambda x. x) @ n 7) \text{ end}
\end{align*}
\]

in which $f$ has type $\Pi \gamma.(\text{int} \rightarrow_\gamma \text{int}) \rightarrow_\gamma \text{int}$ and we have eliminated $\mathbb{N}$.

### 4.3 UVE Specification with Dependent Types

We extend our specification of UVE by introducing two new rules:

\[
\begin{align*}
\Gamma \vdash \lambda x.e : \tau &\Rightarrow m \quad \gamma \not\in \text{FV}(\Gamma) \\
\Gamma \vdash \lambda x.e : \Pi \gamma.\tau &\Rightarrow \lambda \gamma.m \\
\Gamma \vdash e : \Pi \gamma.\tau &\Rightarrow m \\
\Gamma \vdash e @ \tau[a/\gamma] &\Rightarrow m @ a
\end{align*}
\]

The first rule restricts the abstraction of annotation to function expressions. This is the only place in which they are needed. The condition $\gamma \not\in \text{FV}(\Gamma)$ is the expected one, ensuring that we are binding all occurrences of $\gamma$. With these rules added to our specification we can construct a deduction describing the translation given above.
4.4 Correctness

To extend the correctness arguments from the previous section we must first extend the definition of ordering on types to include the following rule:

\[
\tau \leq \tau' \quad \Pi \gamma. \tau \leq \Pi \gamma. \tau'
\]

The statement of Theorem 1 remains the same for stating correctness of UVE with dependent types. The proof can be found in [4].

4.5 Further Examples

Dependent types also prove useful for handling certain recursive functions. Consider the following example [2] (using a more ML-like syntax for readability):

\[
\text{fun } f \ g \ x \ y \ z = \text{if } z \text{ then } (g \ @ x) \\
\quad \text{else } f \ @ (\lambda y.3) \ @ y \ @ x \ @ (z-1)
\]

Assume that \( f \) is used in a context such as

\[
f \ @ (\lambda v. v) \ @ Q_1 \ @ Q_2 \ @ Q_3
\]

in which \( Q_2 \) is effect free. Without using dependent types, our system can only translate the definition of \( f \) to

\[
\text{fun } f \ g \ x \ y \ z = \text{if } z \text{ then } (g \ @_n x) \\
\quad \text{else } f \ @_n (\lambda y.3) \ @_n y \ @_n x \ @_n (z-1)
\]

of type \((\text{int} \rightarrow \text{int}) \rightarrow \text{int} \rightarrow \text{int} \rightarrow \text{int} \rightarrow \text{int} \rightarrow \text{int})\), identifying nothing as unneeded. Using dependent types, however, we can translate the function to

\[
\text{fun } f \ gamma \ g \ x \ y \ z = \text{if } z \text{ then } (g \ @_gamma x) \\
\quad \text{else } f \ @_u (\lambda y.3) \ @_u y \ @_u x \ @_u (z-1)
\]

In this version, we identify the recursive call to \( f \) as having two useless arguments (the actual parameters \( y \) and \( x \)), while identifying the type of its first parameter to be \((\text{int} \rightarrow \text{int})\).

In addition to statically identifying useless code, our system with dependent types supports a form of dynamic identification of useless code. While this code cannot be eliminated statically, it can be ignored at runtime. Consider the example from Section 3 again. That example requires two evaluations of the term \( N \), even though one of them is useless (as an argument to \( f_1 \)). Using UVE with dependent types we can transform that example into the following:

\[
\text{let } f_1 = \lambda x.3 \\
\quad f_2 = \lambda x. x + 1 \\
\quad g = \lambda gamma. \lambda k. (k \ @_gamma N) \\
\quad h = \lambda k. (k \ @_u \ \text{dint}) \\
\quad \text{in } (g \ @_u @_n f_1) + (g \ @_n \ @_n f_2) + (f_1 \ @_u \ \text{dint}) + \\
\quad (h \ @_n f_1) \text{ end}
\]
Now the evaluation of the expression \((g \circ u \circ n \ f1)\) reduces to the expression \(((\lambda x.3) \circ u \circ n)\) and we avoid the useless evaluation of \(N\) in this case. Note that we cannot statically eliminate \(N\) because the expression \((g \circ n \circ n \ f2)\) requires the evaluation of \(N\). This is an example of useless-variable elimination that previous systems could not detect.

Another use of dependent types arises with operands that have an effect. In our specifications without dependent types, an argument to a function may be needed only for its effect. We say that such arguments are relevant, suggesting that their computations are relevant to the behavior of an expression, but not their values. Consider the example

\[
\text{let } f = \mu f.\lambda x.\lambda y.\text{if } z \text{ then } y \text{ else } f \circ (x+1) \circ (y+y)
\]
\[
g \ z = 3
\]
\[
in (g \circ (f \circ a \circ b)) + (g \circ N) \text{ end}
\]

in which \(N\) is an expensive, effect-free expression. Because \(f\) is a recursive function, a reasonable effect analysis will conclude that the expression \((f \circ a \circ b)\) is not effect free (due to the possibility of non-termination). Hence, we cannot consider it a useless argument. It is a relevant argument. Without dependent types, our system considers \(g\) as a function that needs its argument, and hence cannot eliminate \(N\):

\[
\text{let } f = \mu f.\lambda x.\lambda y.\text{if } z \text{ then } y \text{ else } f \circ n \circ (x+1) \circ n \circ (y+y)
\]
\[
g \ z = 3
\]
\[
in (g \circ n \circ (f \circ n \circ a \circ n \circ b)) + (g \circ n \circ N) \text{ end}
\]

With dependent types, we can isolate \(g\)'s need for its argument just to the case in which the argument has an effect:

\[
\text{let } f = \mu f.\lambda x.\lambda y.\text{if } z \text{ then } y \text{ else } f \circ n \circ (x+1) \circ n \circ (y+y)
\]
\[
g \gamma \ z = 3
\]
\[
in (g \circ n \circ n \circ (f \circ n \circ a \circ n \circ b)) + (g \circ u \circ u \circ u \circ d) \text{ end}
\]

The definition of \(g\) here looks a bit strange as we have introduced an apparently useless parameter \(\gamma\). However, the type of \(g\) is \(\Pi \gamma.\text{int} \rightarrow \gamma.\text{int}\).

5 A UVE Algorithm

We have developed and proved correct an algorithm that detects useless code. The algorithm is partitioned into three stages: type inference, constraint generation and constraint solving, defined by the functions \(T\), \(U\), and \(\text{solve}\) respectively.

The first stage is essentially traditional type inference for the simply-typed functional language introduced in Section 2 and so we do not explicitly define it here. Of particular interest, however, is the observation that all function types are annotated. Since no meaning can be attached to these annotations until the constraints are solved, type inference can treat the annotations as variables. The function \(T(e)\) returns a term \(m^\tau\) of type \(\tau\) in which each subterm of \(m\) is explicitly decorated with its inferred type.
The function $U$, defined in Figure 2, takes a term $m^\tau$ and returns a set $\Theta$ of term variables (which is a subset of variables free in $m^\tau$), a set $\Phi$ of constraints on annotations and a term $e'$, which is a form of $m^\tau$ with all applications annotated and all type decorations removed.

Detecting useless variables hinges on the constraint $x \notin \text{FV}(e')$ imposed by the abstraction rule in Figure 1. The set $\text{FV}(e')$ is the set of free variables in the translated term $e'$. In other words, $\text{FV}(e')$ is the set of free variables not detected as useless in the original term. The $U$ function computes a representation of this set of free variables as the set $\Theta$.

In the application rule in Figure 2, the sets $\Theta_1$ and $\Theta_2$ represent the free variables not detected as useless in the operator and operand, respectively. In computing the corresponding set for the application, we must union $\Theta_1$ and $\Theta_2$, provided that the operand is needed. If, on the other hand, the operand is useless, then $\Theta_2$ should not be included in the union.

Since we have no way of determining the usefulness of the operand at this stage of the algorithm, we delay the resolution of the issue by computing the conditional union of $\Theta_1$ and $\Theta_2$. This is done by annotating each term variable in $\Theta_2$ with the annotation variable $\gamma$ (represented in $U$ by the operation $\gamma \Theta_2$). Since several union operations may take place while examining a term, a term variable may be annotated with several different annotation variables. In fact, there may be several occurrences of the same term variable (each with different annotations) within a set. Consider the following example:

$$E \equiv h \text{int} \rightarrow \gamma_1 \text{int} @ [(f \text{int} \rightarrow \gamma_2 \text{int} @ x \text{int}) + (g \text{int} \rightarrow \gamma_3 \text{int} @ x \text{int})]$$

The set $\Theta$ for $E$ is $\{h^\emptyset, f^{\{\gamma_1\}}, g^{\{\gamma_1\}}, x^{\{\gamma_1, \gamma_2\}}, x^{\{\gamma_1, \gamma_3\}}\}$.

If the operand has a side effect, then it must be needed. To enforce this, the constraint $\langle \gamma = n \rangle$ is added to the constraint set. The function $\text{getord}$ returns a set of constraints that enforces the ordering on annotation variables as defined by the subtype relation in Section 3, and is defined as follows:

$$\text{getord}(\text{int}, \text{int}) = \{\}$$

$$\text{getord}(\tau_1 \rightarrow \gamma_1 \tau_2, \tau_1 \rightarrow \gamma_2 \tau_2') = \{\langle \gamma_1 \leq \gamma_2 \rangle^\emptyset\} \cup \text{getord}(\tau_1', \tau_1) \cup \text{getord}(\tau_2, \tau_2')$$

In the application case, the constraint set $\Phi_3$ is conditionally unioned in the same way as the variable set $\Theta_2$. This is justified by the observation that, if the operand is useless, then it should not impose any constraints on the meaning of annotation variables.

In the abstraction rule in Figure 2, the set of free variables not detected as useless is represented by the set $\Theta$ with all occurrences of $x$ removed (represented as the operation $\Theta \setminus x$). The abstraction rule introduces the constraint $\langle x \in \Theta \supset \gamma = n \rangle$. Any variable in $\Theta$ is considered needed while all others are useless. This corresponds to the constraint $x \notin \text{FV}(e')$ in Figure 1.

The variable rule in Figure 2 simply adds an occurrence of $x$ to the set of free variables. The other rules are straightforward.

The third stage of the algorithm solves the constraints generated during the second stage. The function $\text{solve}$, defined in Figure 3, takes a set of constraints
The complete UVE algorithm is defined as follows:

\[
\text{UVE}(e) = \text{let } m^\tau = T(e) \\
(\Theta, \Phi, e') = U(m^\tau) \\
\delta = \text{solve}(\Phi) \\
\text{in } \delta e'
\]

When it is not safe to make assumptions concerning the type of \(e\), all annotation variables in \(\tau\) (as well as annotation variables in the types of all free variables in \(e\)) should be bound to \(n\). To further conform to the specification in Section 3, a trivial translation translate can be defined which replaces all occurrences of \(e_1 \@_U e_2\) in \(e'\) with \(e_1 \@_U d\) where \(d\) is a variable of the same type as \(e_2\).

To prove the correctness of the algorithm, it suffices to show that the algorithm is sound with respect to the specification in Section 3. Before stating the
1. \text{solve}(\{x < \Theta \implies \gamma = n\}^{n} \cup \Delta \cup \Phi) = \text{solve}(\{x < \Theta \implies \gamma = n\}^{\Delta} \cup \Phi)

2. \text{solve}(\{x \in \Theta \implies \gamma = n\}^{\Theta} \cup \Theta \implies \gamma = n\}^{\Theta} \cup \Phi) = \text{solve}(\{x \in \Theta \implies \gamma = n\}^{\Theta} \cup \Phi)

3. \text{solve}(\{x \in \Delta \cup \Phi\} \cup \Phi) = \text{solve}(\{n \in \Delta \cup \Phi\})

4. \text{solve}(\{\gamma = n\}^{n} \cup \Delta \cup \Phi) = \text{solve}(\{\gamma = n\}^{\Delta} \cup \Phi)

5. \text{solve}(\{\gamma = n\}^{\Theta} \cup \Phi) = \text{solve}(\{\Phi[n/\gamma]\} \circ \{\gamma \mapsto n\})

6. \text{solve}(\{n \in \Delta \cup \Phi\} = \text{solve}(\Phi)

7. \text{solve}(\{\gamma_1 \leq \gamma_2\}^{n} \cup \Delta \cup \Phi) = \text{solve}(\{\gamma_1 \leq \gamma_2\}^{\Delta} \cup \Phi)

8. \text{solve}(\{\gamma \leq n\}^{\Theta} \cup \Phi) = \text{solve}(\Phi[n/\gamma]) \circ \{\gamma \mapsto n\})

9. \text{solve}(\{\gamma \leq n\}^{\Theta} \cup \Phi) = \text{solve}(\Phi)

10. \text{solve}(\{n \leq \gamma\}^{\Theta} \cup \Phi) = \text{solve}(\Phi)

11. otherwise, \text{solve}(\Phi) = \delta_u

\textbf{Fig. 3. Solving constraints}

Theorem, we require a few auxiliary lemmas. The first two state the correctness of the constraint solver.

\textbf{Lemma 1.} Given a finite set of constraints \(\Phi\), \text{solve}(\Phi) always terminates.

\textbf{Lemma 2.} If \text{solve}(\Phi) = \delta for any constraint set \(\Phi\), then constraint \(c\) is satisfied for all \(c \in \delta\Phi\).

The third lemma guarantees that the constraint set generated by the function \text{getord} enforces the type ordering defined in Section 3.

\textbf{Lemma 3.} If \text{getord}(\tau_1, \tau_2) = \Phi, \Phi \subseteq \Phi', and \text{solve}(\Phi') = \delta for any types \(\tau_1\) and \(\tau_2\), and constraint set \(\Phi'\), then \(\delta \tau_1 \leq \delta \tau_2\).

We also require the following two definitions:

\textbf{Definition 3.} The term \(|m\tau|\) is the term \(m\) with all type decorations removed.

\textbf{Definition 4.} The function \text{simplify}(\Theta) returns the set of variables \(x_i\) such that \(x_i^{\Delta} \in \Theta\) and \(\Delta\) contains no occurrences of \(u\).

The following theorem states the correctness of the UVE algorithm by proving the algorithm’s soundness with respect to the type system in Section 3.

\textbf{Theorem 2.} If \(U(m^\tau) = (\Theta, \Phi, e), \Phi \subseteq \Phi', \text{solve}(\Phi') = \delta, \) and \text{translate}(\delta e) = e' for any well-typed, annotated term \(m^\tau\) and constraint set \(\Phi'\), then

1. \(FV(e') = \text{simplify}(\delta \Theta)\)
2. \(\Gamma \vdash |m^\tau| : \delta \tau \Rightarrow e'\), for all \(\Gamma\) such that \(\text{dom}(\Gamma) \supseteq FV(m^\tau)\) and \(\Gamma(x) = \delta \tau'\) for all \(x^{\tau'}\) free in \(m^\tau\).
Part 1 of the theorem states that the algorithm computes the set of free variables in the translated term, which is required in Part 2 to prove that the algorithm imposes the same constraints on annotations as the inference system in Section 3. We make use of Lemma 2 to assume that we have a solution to the set $\Phi'$, from which we can assume that we have a solution to the subset $\Phi$.

The proofs of both parts of the theorem as well as the auxiliary lemmas can be found in [4].

6 Conclusions

We have presented a specification of useless-variable and useless-code elimination for a higher-order, call-by-value language and shown it to be both sound and safe. By using dependent types we can identify code that, while not statically eliminated, can be ignored in some instances at run time. We have presented a simple algorithm based on our specification without dependent types and shown the algorithm to be sound with respect to our specification. We are working on an algorithm for UVE with dependent types that introduces a minimal number of annotation-variable abstractions.

Acknowledgements. We thank the referees for their comments and suggestions. We also thank Naoki Kobayashi for his comments on an earlier draft of this paper.

References

Boolean Constraints for Binding-Time Analysis

Kevin Glynn, Peter J. Stuckey, Martin Sulzmann, and Harald Søndergaard

Department of Computer Science and Software Engineering,
The University of Melbourne, Victoria 3010, Australia
{keving, pjs, sulzmann, harald}@cs.mu.oz.au

Abstract. To achieve acceptable accuracy, many program analyses for functional programs are “property polymorphic”. That is, they can infer different input-output relations for a function at separate applications of the function, in a manner similar to type inference for a polymorphic language. We extend a property polymorphic (or “polyvariant”) method for binding-time analysis, due to Dussart, Henglein, and Mossin, so that it applies to languages with ML-style type polymorphism. The extension is non-trivial and we have implemented it for Haskell. While we follow others in specifying the analysis as a non-standard type inference, we argue that it should be realised through a translation into the well-understood domain of Boolean constraints. The expressiveness offered by Boolean constraints opens the way for smooth extensions to sophisticated language features and it allows for more accurate analysis.

1 Introduction

The aim of this paper is to assimilate sophisticated program analysis capabilities in the context of a modern functional programming language with structured data and ML-style polymorphism. The most important capability of interest is “property polymorphism”, that is, an analyser’s ability to infer different properties for a definition \( f \) at separate uses of \( f \), in a manner similar to type inference for a polymorphic language. For many program analyses, we need property polymorphism to achieve acceptable accuracy in the analysis. A second feature that we want is modularity of analysis, that is, the ability to produce analysis results that are context-independent, so as to support separate compilation.

“Property polymorphism” has previously been studied for a variety of program analyses for higher-order functional programs. However, usually, the underlying language misses features such as algebraic types and polymorphism. The assumption is usually that an extension, for example to ML-style polymorphism, is straightforward. Recent work, however, suggests that this is not necessarily so [15, 27]. One aim of this paper is to show that Boolean constraints give a better handle on such extensions.

In this work we take binding-time analysis as our example. This analysis is used for program specialisation. The purpose of binding-time analysis is to identify program expressions that can be evaluated at specialisation time, based on information about which parts of the program’s input will be available at
that time. Values that are known at specialisation-time are called *static*, while values that will only be known later are called *dynamic*.

Our interest in binding-time analysis is mainly as an example of a non-trivial program analysis in functional programming. More generally we are interested in program analysis expressed as inference of “constrained types”. In Section 8 we discuss related analyses to which our methodology also applies.

In the context of binding-time analysis, “binding-time polymorphism” is often referred to as *polyvariance*. Our analysis is polyvariant and extends the analysis proposed by Dussart, Henglein and Mossin [7] to polymorphically typed programs. To investigate our method’s practicality, we have implemented our binding-time analysis for the Glasgow Haskell Compiler (GHC). This assigns binding-time properties to all expressions, but we do not have a specialiser.

Much work on type-based analysis of functional programs introduces new constraint systems to define properties of interest, proceeding to explain how constraint solving or simplification is done for the new system. In many cases, to obtain accurate analysis, it appears necessary to employ non-standard versions of esoteric type systems such as intersection types.

We deviate from this path by deliberately seeking to utilise well-known constraint domains of great expressiveness. There are advantages of using well-known constraint domains:

- Useful known theorems can be used to simplify the presentation or improve the algorithms. For example, much is known about the complexity of constraint solving for various fragments of propositional logic.
- Extra expressiveness may assist extensions of the analysis to new language features. In Section 6 we argue that the extension to ML-style polymorphism is facilitated by the constraint view, and claim that this view helps us navigate the design space for the analysis.
- Extra expressiveness may allow for more accurate analysis. This is borne out by experience from other other programming language paradigms [2].
- An implementation may utilise efficient representations and algorithms. (For this paper, we have experimented with two different Boolean solvers.)

In the binding-time analysis literature we find many examples of monomorphic analysis for a polymorphic language, for example Mogensen’s analysis [16], as well as polyvariant analysis for a monomorphic language [1]. The situation is similar for other analyses; for example, recent progress in usage analysis deals with either usage-monomorphic analysis for a polymorphic language [27] or usage-polymorphic analysis for a monomorphic language [28]. To our knowledge, this is the first time polyvariant binding-time analysis has been developed and implemented for a polymorphic language. Also, to the best of our knowledge, we present the first formalisation of the underlying binding-time logic. Owing to limited space, we concentrate on the core of our analysis and its novel aspects. For example, we leave out the treatment of structured data, focusing on basic types and function types.

The next section introduces basic concepts concerning types and constraints. Section 3 introduces the binding-time constraint system and logic. Section 4
shows how to translate binding-time constraints to Boolean constraints. In Section 5 we extend the binding-time inference system of Dussart, Henglein and Mossin to polymorphically typed programs. The use of Boolean constraints provides us with a large design space. In Section 6 we describe alternative methods for supporting polymorphic function application. In Section 7 we discuss a method for finding fixed points by adding constraints. Section 8 discusses an implementation and concludes.

2 Preliminaries

2.1 The Underlying Type System

Programs are assumed to be well-typed in an underlying type system which we will refer to as UL. We consider an ML-style let-polymorphic typed language with base types Int and Bool:

Types \( t ::= \alpha \mid \text{Int} \mid \text{Bool} \mid t \rightarrow t \)

Type Schemes \( \sigma ::= t \mid \forall \bar{\alpha}.t \)

Expressions \( e ::= x \mid \lambda x.e \mid ee \mid \text{let } x = e \text{ in } e \mid x\sharp \bar{t} \)

Expressions also include numerals and Boolean values True and False. We use vector notation for sequences. For example, \( \bar{\alpha} \) represents a sequence \( \alpha_1, \ldots, \alpha_n \) of type variables. Types in UL are referred to as underlying types.

We assume well-typed expressions to be explicitly typed. An example well-formed, typed expression is

\[
\text{let } f = \lambda x.x :: \forall \alpha.\alpha \rightarrow \alpha \text{ in } ((f \sharp \text{Int}) 1 :: \text{Int}, (f \sharp \text{Bool}) \text{True} :: \text{Bool})
\]

We find it helpful to have polymorphic application (denoted by \( \sharp \)) explicit in the language, but we let polymorphic abstraction be implicit (restricted to let-statements), to avoid cluttering expressions.

2.2 Binding-Time Types

In line with Dussart, Henglein and Mossin, we impose a “binding-time type” structure on top of the underlying type structure. For instance, \( S \mapsto D \) describes a static function that takes a static value of base type as an argument and returns a dynamic value of base type. The structure of binding-time types reflects the structure of the underlying type system.

Annotations \( b ::= \delta \mid S \mid D \)

Binding-Time Types \( \tau ::= \beta \mid b \mid \tau \vdash b \tau \)

Binding-Time Type Schemes \( \eta ::= \tau \mid \forall \beta, \delta. C \Rightarrow \tau \)

Note that we distinguish between annotation variables \( \delta \) (which may only be instantiated to \( S \) or \( D \)) and binding-time type variables \( \beta \) (which may be instantiated to any \( \tau \), including \( \delta \)). In practice it is not necessary for an implementation to distinguish the two, but for clarity we will do so in this presentation. We write \( \bar{\beta}, \bar{\delta} = fv(\eta) \) to refer to the free binding-time and annotation variables in \( \eta \). We now describe the constraint component \( C \) in binding-time type schemes \( \eta \).
(Sta) \( C \vdash (S \leq_a b) \) \hspace{1cm} (Dyn) \( C \vdash (b \leq_a D) \)

(Hyp) \( C_1, (b_1 \leq_a b_2), C_2 \vdash (b_1 \leq_a b_2) \)

(Refl) \( C \vdash (b \leq_a b) \) \hspace{1cm} (Trans) \( \frac{C \vdash (b_1 \leq_a b_2) \quad C \vdash (b_2 \leq_a b_3)}{C \vdash (b_1 \leq_a b_3)} \)

(Bas\(_w\)) \( C \vdash \text{wft}(b) \) \hspace{1cm} (Arrow\(_w\)) \( \frac{C \vdash (b_1 \leq_f \tau_1) \quad C \vdash \text{wft}(\tau_1)}{C \vdash \text{wft}(b_1 \mapsto \tau_2)} \)

(Bas\(_f\)) \( \frac{C \vdash (b_1 \leq_a b_2)}{C \vdash (b_1 \leq_f b_2)} \) \hspace{1cm} (Arrow\(_f\)) \( \frac{C \vdash (b_1 \leq_a b_2)}{C \vdash (b_1 \leq_f \tau_1 \mapsto \tau_3)} \)

(Bas\(_s\)) \( \frac{C \vdash (b_1 \leq_a b_2)}{C \vdash (b_1 \leq_s b_2)} \) \hspace{1cm} (Arrow\(_s\)) \( \frac{C \vdash (b_2 \leq_a b_3)}{C \vdash (b_2 \leq_s \tau_1 \mapsto \tau_3 \leq_s \tau_4 \mapsto \tau_6)} \)

\( C \vdash (\tau \leq \tau) \mid \text{wft}(\tau) \mid C \land C \mid \exists \bar{\beta}, \bar{\delta}.C \)

For binding-time types it is necessary to add an additional kind of constraint, that binding-time types are “well-formed”. If the top-most annotation of a binding-time type is \textit{dynamic} then we know nothing of its components and so they must all be \textit{dynamic} too. For instance, \( S \vdash^D S \) is meaningless. The rules for constraints \( \text{wft}(\cdot) \) and \( (\cdot \leq_f \cdot) \) ensure that binding-time types are well-formed. Relation \( (b_1 \leq_f \tau) \) ensures that \( b_1 \) is smaller than the top-most annotation of \( \tau \). For monomorphic underlying types, all constraints are reducible to \((\cdot \leq_a \cdot)\) constraints.

Figure 1 defines the primitive fragment of the binding-time constraint system, \textbf{BTC}. This is identical to the rules presented by Dussart, Henglein and
Mossin. We differ, however, by explicitly allowing for existential quantification over binding-time type variables in constraints. The rules governing the existential quantification are taken from cylindric constraint systems [21]. That is,

1. \( C \vdash \exists \vec{\beta}, \vec{\delta}.C \)
2. if \( C_1 \vdash C_2 \) then \( \exists \vec{\beta}, \vec{\delta}.C_1 \vdash \exists \vec{\beta}, \vec{\delta}.C_2 \)
3. \( \exists \vec{\beta}, \vec{\delta}.(C_1 \land \exists \vec{\beta}, \vec{\delta}.C_2) \equiv \exists \vec{\beta}, \vec{\delta}.C_1 \land \exists \vec{\beta}, \vec{\delta}.C_2 \)
4. \( \exists \vec{\beta}_1, \vec{\beta}_2, \vec{\delta}_1, \vec{\delta}_2, C \equiv \exists \vec{\beta}_2, \vec{\delta}_2, \exists \vec{\beta}_1, \vec{\delta}_1, C \)

The purpose of the cylindrification operation \( \exists \vec{\beta}, \vec{\delta} \) is to hide (or discard) the variables \( \vec{\beta}, \vec{\delta} \). During inference of binding-time types we may introduce intermediate variables, not free in the resulting binding-time types or environments. At certain stages during inference, constraints that have been imposed on these variables will no longer be of interest, and the cylindrification operator can then be employed to simplify constraints. Inference will in fact be using Boolean constraints. When we later translate binding-time constraints to a Boolean form, the existential quantification will turn into existential quantification over Boolean variables.

**Example 1.** The identity function \( \lambda x.x \) has the polymorphic type \( \forall \alpha.\alpha \rightarrow \alpha \). Its binding-time type is \( \forall \beta_1, \beta_2, \delta. (\beta_1 \leq \beta_2) \land (\delta \leq \beta_1) \land (\delta \leq \beta_2) \Rightarrow \beta_1 \rightarrow \beta_2 \) where the constraint \( (\beta_1 \leq \beta_2) \land (\delta \leq \beta_1) \land (\delta \leq \beta_2) \) describes the binding-time behaviour of the identity function. The first conjunct says that output from the function is dynamic if input is. The other two conjuncts express a well-formedness condition. Note that the binding-time type variables \( \beta_1 \) and \( \beta_2 \) must range over the same type structure as variable \( \alpha \). An instance of \( \lambda x.x \)'s type is \( (\text{Int} \rightarrow \text{Int}) \rightarrow (\text{Int} \rightarrow \text{Int}) \) with

\[
(D \xrightarrow{S} S) \xrightarrow{S} (D \xrightarrow{S} S)
\]

a possible binding-time type. This expresses that \( \lambda x.x \) can take a function of binding-time type \( D \xrightarrow{S} S \) and the application’s context can treat the result as being of the same binding-time type. Another possible binding-time type is

\[
(D \xrightarrow{S} S) \xrightarrow{S} (D \xrightarrow{D} D)
\]

showing how the binding-time type system allows for coercion: It is acceptable for a static value to be treated as dynamic, if required by the context.

### 2.4 Shapes

To relate an expression’s valid binding-time type to its underlying type, we employ a “shape” system. In Example 1 the identity function’s binding-time type \( \forall \beta_1, \beta_2, \delta. C \Rightarrow \beta_1 \rightarrow \beta_2 \) has shape \( \forall \alpha.\alpha \rightarrow \alpha \). Formally, a shape environment \( \Delta \) maps a polymorphic binding-time type variable to its corresponding underlying polymorphic type variable. If \( \Delta = \{ \beta_{11} : \alpha_1, \ldots, \beta_{nk} : \alpha_n \} \) then \( \text{domain}(\Delta) = \beta \)
\[(\text{Bool}_\delta) \; \Delta \vdash \delta : \text{Bool} \quad (\text{Int}_\delta) \; \Delta \vdash \delta : \text{Int} \]

\[(\text{Bool}_S) \; \Delta \vdash S : \text{Bool} \quad (\text{Bool}_D) \; \Delta \vdash D : \text{Bool} \]

\[(\text{Int}_S) \; \Delta \vdash S : \text{Int} \quad (\text{Int}_D) \; \Delta \vdash D : \text{Int} \]

\[(\beta) \; (\beta : \alpha) \in \Delta \implies \Delta \vdash \beta : \alpha \quad \text{(Arrow)} \quad \frac{\Delta_1 \vdash \tau_1 : t_1 \quad \Delta_2 \vdash \tau_2 : t_2}{\Delta_1 \cup \Delta_2 \vdash \tau_1 \overset{b}{\mapsto} \tau_2 : t_1 \rightarrow t_2} \]

\[\Delta' = \{\beta_{11} : \alpha_1, \ldots, \beta_{k_1} : \alpha_1, \ldots, \beta_{n} : \alpha_n\} \]

\[\Delta \vdash \forall \overline{\beta}, \overline{\delta}. C \Rightarrow \tau : \forall \overline{\alpha}. t \]

**Fig. 2.** Shape rules

is the *domain* of \(\Delta\), and \(\text{range}(\Delta) = \overline{\alpha}\) is the *range*. The judgement \(\Delta \vdash \eta : \sigma\) states that under shape environment \(\Delta\) the binding-time type \(\eta\) has shape \(\sigma\). A judgement \(\Delta \vdash \eta : \sigma\) is valid if it can be derived by the shape rules in Figure 2.

For simplicity, we usually omit information such as \(\{\beta_1 : \alpha, \beta_2 : \alpha\}\) in a type scheme like \(\forall \beta_1, \beta_2, \delta. (\beta_1 \leq \beta_2) \land (\delta \leq \beta_1) \land (\delta \leq \beta_2) \Rightarrow \beta_1 \overset{\delta}{\mapsto} \beta_2\). Shape information is easily recovered by inspection of an expression’s underlying type and the corresponding binding-time type.

Shape inference will be important when performing binding-time analysis. Given an underlying type \(t\), we will sometimes want the most general shape environment \(\Delta\) and binding-time type \(\tau\) such that \(\Delta \vdash \tau : t\). We say \((\Delta, \tau)\) is more general than \((\Delta', \tau')\) iff \(\Delta \vdash \tau : t\), \(\Delta' \vdash \tau' : t\) and there exists a substitution \(\phi\) such that \(\phi \Delta = \Delta'\) and \(\phi \tau = \tau'\).

**Lemma 1.** Given a type \(t\), one can compute the most general shape environment \(\Delta\) and binding-time type \(\tau\) such that \(\Delta \vdash \tau : t\).

Lemma 1 ensures that there exists an algorithm which computes the most general shape environment \(\Delta\) and binding-time type \(\tau\) given an underlying type \(t\). In our presentation, we treat the algorithm as a deduction system with clauses of the form \(t \vdash (\tau, \Delta)\) where \(t\) is the input value and \(\Delta\) and \(\tau\) are the output values. Similarly, there is an algorithm which computes the most general binding-time type \(\tau\), given a shape environment \(\Delta\) and an underlying type \(t\) (we write \(\Delta, t \vdash \tau\)). Finally, there is an algorithm to compute the most general shape environment \(\Delta\) given a binding-time type \(\tau\) and an underlying type \(t\) (we write \(\tau, t \vdash \Delta\)).
3 Binding-Time Logic

We assume expressions to be well-typed in the underlying type system. Binding-time properties of expressions are specified by typing judgements which are of the form $C, \Gamma \vdash (e :: \sigma) : \eta$ where $C$ is in BTC, $\Gamma$ is a binding-time type environment assigning binding-time types to the free variables of $e$, $(e :: \sigma)$ is a type-annotated expression, and $\eta$ is $e$’s binding-time type. Always implicit, is a shape environment $\Delta$ associating binding-time type variables with underlying type variables. We assume that we can translate a program’s initial type environment (which maps primitive functions to their underlying types), into an initial binding-time environment mapping primitive functions to their binding-time types. We require that the binding-time type of a primitive function is “directed”:

**Definition 1 (Well-shapedness, Polarities, Directedness).** Given a type scheme $\sigma = \forall \bar{\alpha} . t$, a binding-time type $\eta = \forall \bar{\beta} . C \Rightarrow \tau$, a shape environment $\Delta$, a constraint $C'$, and a binding-time type $\tau'$.

- We say that $\eta$ is well-shaped (wrt. $\sigma$ and $\Delta$) iff $\Delta \vdash \eta : \sigma$.
- We say that $(C', \eta)$ is well-shaped iff $\eta$ is well-shaped.
- We define polarities recursively: $\beta$ appears in positive position in $\tau'$ (written $\tau'[\beta^+]$) iff $\tau' = \tau_1 \mapsto \tau_2$ and either $\tau_1[\beta^-]$ or $\tau_2[\beta^+]$. Similarly $\beta$ appears in negative position in $\tau'$ (written $\tau'[\beta^-]$) iff $\tau' = \tau_1 \mapsto \tau_2$ and $\tau_1[\beta^+]$ or $\tau_2[\beta^-]$.
- We say that $(C', \eta)$ is directed (wrt. $\sigma$ and $\Delta$) iff (1) $\eta$ is well-shaped, and (2) there exists an extension $\Delta' \supseteq \Delta$ such that $\Delta' \vdash \tau : t$ and (3) for each $\beta_1, \beta_2$ such that $C \land C' \vdash (\beta_1 \leq \alpha \leq s \beta_2)$, (a) $\Delta' \vdash \beta_1 : \alpha$ and $\Delta' \vdash \beta_2 : \alpha$ (that is, $\beta_1$ and $\beta_2$ have same type according to $\Delta'$) and (b) $\tau[\beta_1^+]$ and $\tau[\beta_2^-]$ (that is, $\beta_1$ occurs in negative position and $\beta_2$ occurs in positive position).
- We say that $\eta$ is directed iff $(true, \eta)$ is directed.

These definitions extend to type environments in the natural way.

The directedness condition simply states that all constraints of the form $(\beta_1 \leq \alpha \leq \beta_2)$ describe relations between input values $\beta_1$ and output values $\beta_2$. This is naturally fulfilled by a binding-time analysis, since intuitively, information flows from negative to positive positions.

**Example 2.** A directed binding-time description of if–then–else ($ite :: \text{Bool} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$) is as follows:

$$ite : \forall \delta, \beta_1, \beta_2, \beta_3, (\beta_1 \leq s \beta_3) \land (\beta_2 \leq s \beta_3) \land (\delta \leq f \beta_3) \Rightarrow \delta \mapsto \beta_1 \mapsto \beta_2 \mapsto \beta_3$$

Note that the constraint $(\delta \leq f \beta_3)$ vacuously satisfies the directedness condition. The constraints $C$ that can appear in type schemes and on the left hand side of the turnstile are restricted to those generated by the grammar:

$$C ::= (b_1 \leq a \ b_2) \mid (\tau_1 \leq s \ \tau_2) \mid (b \leq f \ \tau) \mid \text{wft}(\tau) \mid C \land C' \mid \exists \bar{\beta}, \delta, C$$
where $\tau_1$ and $\tau_2$ must have the same shape. This excludes, for example, constraints of the form $(\beta \leq_s S \mapsto D)$. Such a restriction is necessary, as we do not allow for subtyping in the underlying type system $UL$.

Figure 3 defines the binding-time logic. $\Gamma_x$ denotes the environment $\{y : \eta \in \Gamma \mid y \neq x\}$. Most of the rules are straightforward. For example, rule (Sub) expresses the obvious subsumption (or coercion) rule for binding-time types. Rule (Abs) says that, in order to deduce the binding-time type $\tau_1 \mapsto \tau_2$ for an expression $(\lambda x.e :: t_1 \rightarrow t_2)$, we must be able to deduce $\tau_2$ for $(e :: t_2)$ under
the additional assumption that $x$ has type $\tau_1$ (which is well-formed and of shape $t_1$). Rule (∃I) allows unnecessary binding-time type variables in the constraint component of a typing judgement to be discarded. In rule (∀I), we require that quantification over $\beta$ variables is restricted to $\beta$s related to polymorphic variables $\alpha$ by the shape environment.

Rule (∀E) allows instantiation of binding-time type schemes, at a variable’s usage site. We need to compute the corresponding binding-time type instances for each underlying type instance. Let $\Delta$ be a shape environment, $\bar{t}$ a sequence of underlying types, and $\bar{\alpha}$ a sequence of underlying type variables. We define $\text{inst}(\Delta, \bar{t}, \bar{\alpha}) = \bar{\tau}$ where $\bar{\tau}$ are fresh binding-time types of appropriate shape: Each element of $\bar{\tau}$ is related to the corresponding element of $\bar{\beta}$ by the shape environment $\Delta$. That is, $\Delta, t_i \vdash \tau_{ij}$ where $\Delta \vdash \beta_{ij} : \alpha_i$.

A judgement $C, \Gamma \vdash (e :: \sigma) : \eta$ is valid if the judgement is derivable by the rules in Figure 3.

**Lemma 2 (Conservative Extension).** Let $e$ be well-typed with type $\sigma$ and let the binding-time environment $\Gamma$ be well-formed and directed. Then $C, \Gamma \vdash (e :: \sigma) : \eta$ for some constraint $C$ and binding-time type $\eta$.

From here on we consider only expressions that are well-typed in UL.

The following lemma allows us to ignore well-formedness constraints in examples. We will also assume from now on that all binding-time types are directed.

**Lemma 3 (Well-Formed and Directed).** Let $e$ be well-typed with type $\sigma$. If $C, \Gamma \vdash (e :: \sigma) : \eta$ and $\Gamma$ is well-formed and directed, then $(C, \eta)$ is well-formed and directed.

We define an ordering on binding-time type schemes for use in the inference algorithm (read $C'' \vdash \eta \leq \eta'$ as $\eta$ is more general than $\eta'$ in the context of constraint $C''$):

$$C'' \vdash (\forall \bar{\beta}, \bar{\delta}.C \Rightarrow \tau) \leq (\forall \bar{\beta}', \bar{\delta}'.C' \Rightarrow \tau') \iff C'' \wedge C' \vdash \exists \bar{\beta}, \bar{\delta}.C \wedge (\tau \leq_s \tau').$$

(We assume here, with no loss of generality, that there are no name clashes.)

## 4 Translation to Boolean Constraints

Ultimately, binding-time constraints are nothing more than Boolean constraints. We can read the binding-time values $S$ and $D$ as false and true, respectively. $\delta$ variables are Boolean variables, and a constraint $(\delta_1 \leq_a \delta_2)$ is simply an implication $\delta_1 \rightarrow \delta_2$. Since all valid binding-time type constraints $(\tau_1 \leq_s \tau_2)$ have the same shape, every binding-time constraint can also be understood as a Boolean constraint. We map a binding-time constraint $C$ to its corresponding Boolean constraint $\llbracket C \rrbracket$ as follows:
We note that $C \vdash C'$ iff $[C] \models [C']$, where $\models$ denotes entailment in propositional logic.

Since the class of Boolean constraints generated is a subclass of the set HORN of propositional Horn clauses, an immediate advantage of the translation is that we have linear time algorithms for satisfiability [6], and hence many other operations. If more sophisticated analysis calls for a larger class of Boolean constraints then there are efficient representations and algorithms for Boolean constraints available, for example based on ROBDDs [3]. Finally, useful operations such as existential and universal quantification, conjunction and disjunction have a clear and well-understood meaning.

5 Binding-Time Inference

We assume that we are given a well-typed program, each subexpression annotated with its underlying type. Binding-time inference computes the missing binding-time information. The inference algorithm, Figure 4, is formulated as a deduction system over clauses of the form

$$\Gamma, e :: t \vdash_{inf} (C, \tau)$$

with a binding-time type environment $\Gamma$ and a type-annotated expression $e$ as input, and a binding-time constraint $C$ and a binding-time type $\tau$ as output. An algorithm in style of algorithm $W$ can be derived from this given specification in a straightforward way.

All rules are syntax-directed except rule ($\exists$ Intro). This rule is justified by the corresponding rule in the logical system. We assume that rule ($\exists$ Intro) is applied aggressively, so that useless variables do not appear in analysis results. Rule (Var-λ) handles lambda-bound variables whereas rule (Var–Inst) handles instantiation of let-bound variables. Considering the binding-time logic of Figure 3 the straightforward approach to polymorphism is to instantiate polymorphic binding-time variables in both constraints and binding-time types with a $\tau$ of the appropriate shape.
Example 3. Consider the (type annotated) polymorphic application

\[(\text{id} : \text{Int} \rightarrow \text{Int}) :: (\text{Int} \rightarrow \text{Int}) \rightarrow (\text{Int} \rightarrow \text{Int})\] \[(g : \text{Int} \rightarrow \text{Int}) :: \text{Int} \rightarrow \text{Int}\]

where \(\text{id} :: \forall \alpha. \alpha \rightarrow \alpha\). Here are binding-time descriptions for \(\text{id}\), \(g\) and \((\text{id} \; g)\):

\[
\begin{align*}
\text{id} & : ((\beta_1 \leq_s \beta_3) \land (\delta_2 \leq_f \beta_1) \land (\delta_2 \leq_f \beta_3), \beta_1 \overset{\delta_2}{\rightarrow} \beta_3) \\
g & : ((\delta_5 \leq_a \delta_4) \land (\delta_5 \leq_a \delta_6), \delta_4 \overset{\delta_5}{\rightarrow} \delta_6) \\
id \; g & : \delta_7 \overset{\delta_8}{\rightarrow} \delta_9
\end{align*}
\]
We need to build an instance of \textit{id} where all polymorphic binding-time variables of shape \( \alpha \) are instantiated to binding-time types of shape \((\texttt{Int} \to \texttt{Int})\). We substitute \( \delta_{10} \mapsto \delta_{12} \) for \( \beta_1 \), \( \delta_{14} \mapsto \delta_{16} \) for \( \beta_3 \) and \( \delta_{13} \) for \( \delta_2 \), resulting in

\[(\textit{id} \# \texttt{Int} \to \texttt{Int})::(\texttt{Int} \to \texttt{Int}) \to (\texttt{Int} \to \texttt{Int}) : (\delta_{10} \mapsto \delta_{12}) \mapsto (\delta_{14} \mapsto \delta_{16})\]

under the “instantiated” constraints

\[((\delta_{10} \mapsto \delta_{12}) \leq_s (\delta_{14} \mapsto \delta_{16})) \wedge (\delta_{13} \leq_f (\delta_{10} \mapsto \delta_{12})) \wedge (\delta_{13} \leq_f (\delta_{14} \mapsto \delta_{16}))\]

These constraints are translated into Boolean constraints using the constraint rules of Figure 1 and Section 4. We then proceed to analyse \((\textit{id} g)\) by applying the usual inference rules.

Rule (Let) introduces annotation polymorphism as well as type polymorphism. Note that we can not simply generalise over all the free binding-time type variables because the expression might not be annotated with its principal type.

**Example 4.** Consider the following type-annotated program:

\[
g :: \forall \alpha_1.\alpha_1 \to ((\alpha_1, \texttt{Int}), (\alpha_1, \texttt{Bool}))
g x = \text{let } f :: \forall \alpha_2.\alpha_1 \to \alpha_2 \to (\alpha_1, \alpha_2)
\quad f y z = (y, z)
\quad \text{in } ((f \# \texttt{Int}) x 1, (f \# \texttt{Bool}) x \text{True})
\]

Whatever binding-time variable \( \beta \) we assign to \( y \), we are not allowed to quantify over \( \beta \).

Therefore a polymorphic binding-time variable is only considered free if its corresponding polymorphic type variable is free.

We define a generalisation function giving the generalised type scheme and the generalised constraint. Let \( C \) be a constraint, \( \Gamma \) a type environment, \( \tau \) a binding-time type, \( t \) an underlying type and \( \bar{\alpha} \) a sequence of underlying type variables. Then

\[\text{gen}(C, \Gamma, \tau, t, \bar{\alpha}) = (\exists \bar{\beta}, \bar{\delta}. C, \forall \bar{\beta}, \bar{\delta}. C \Rightarrow \tau)\]

where

- \( \tau, t \vdash \Delta \)
- \( \bar{\beta} = \{ \beta \mid \Delta \vdash \beta : \alpha \text{ for some } \alpha \in \bar{\alpha} \} \)
- \( \bar{\delta} = \text{fv}(C, \tau) \setminus (\text{fv}(\Gamma) \cup \text{domain}(\Delta)) \)

We first associate all free binding-time type variables with all free underlying type variables. Then, a binding-time variable \( \beta \) is free if the corresponding type variable \( \alpha \) is free. It remains to compute all free annotation variables \( \delta \). We further note that we push the whole constraint \( C \) into the type scheme (one could be more efficient by pushing in only the affected constraints).
In rules (Var-λ), (Abs), (App), and (Fix), given an underlying type $t$, we generate a most general binding-time type $\tau$ (the most general shape environment $\Delta$ is not of interest) such that $\Delta \vdash t : \tau$. Moreover, we add the constraint $wft(\tau)$ to the output constraint to ensure well-formedness of the newly generated binding-time type. For example, a $\lambda$-bound variable of type $\text{Int} \rightarrow \text{Int}$, could give rise to a binding-time type of $\delta_1 \delta_2 \rightarrow \delta_3$. The constraint $wft(\delta_1 \delta_2 \rightarrow \delta_3)$ reduces to $\delta_2 \rightarrow \delta_1 \land \delta_2 \rightarrow \delta_3$.

With (Fix), we follow Dussart, Henglein and Mossin [7], performing Kleene-Mycroft iteration until a fixed point is found. Define

$$F(\Gamma_x.x : \eta_i, e :: t) = F(\Gamma_x.x : \eta_{i+1}, e :: t) \text{ if } \text{true} \vdash \eta_i < \eta_{i+1}$$

where $\Gamma_x.x : \eta_i, e :: t \vdash_{inf} (C, \tau)$ and $(\cdot, \eta_{i+1}) = \text{gen}(C, \Gamma_x.x : \eta_i, \tau)$.

Note that the sequence $\eta_0 \leq \ldots \leq \eta_i \leq \ldots$ is necessarily finite: The binding-time constraint rules (Figure 1) ensure that only binding-time types of the same shape are comparable. Moreover, as the generated binding-time type schemes are of the form $\forall \delta.C \Rightarrow \tau$, that is, the quantifier is over annotation variables only (not binding-time type variables), each only has a finite number of instances.

**Example 5.** Consider the binding-time analysis of the following program

$$g :: \text{Bool} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow (\text{Int}, \text{Int})$$

$$g \ p \ x \ y = \text{if } p \text{ then } (x,y) \text{ else } (\text{snd } (g \ p \ y \ x), y)$$

or, in our syntax,

$$\text{fix } g \text{ in } \lambda p. \lambda x. \lambda y. \text{ite } p \ (x,y) \ (\text{snd } (g \ p \ y \ x), y)$$

where

- $(e_1, e_2)$ builds a pair from two expressions.
- $\text{snd}$ returns the second element of a pair.
- $\text{ite}$ is a built-in if–then–else function which returns a pair of Ints. (Two extensions of the algorithm, supported by our implementation, allow a more natural definition: (1) If–then–else can be written using a built-in case expression, and (2) polymorphic application allows us to use the usual if–then–else function and instantiate it to the pair in this context.)

Fixed point calculation starts with a fresh binding-time type scheme $\eta_0$ for $g$:

$$g : \forall \delta.C \Rightarrow \delta_1 \delta_2 \delta_8 \delta_9 \delta_4 \delta_5 \delta_6$$

In terms of Boolean constraints, the binding-time type scheme ordering given at the end of Section 4 translates to: $(\forall \delta_1.C_1 \Rightarrow \tau_1) \leq (\forall \delta_2.C_2 \Rightarrow \tau_2)$ iff $C_2 \vdash \exists \delta_1.(C_1 \land (\tau_1 \leq \tau_2))$. (Without loss of generality we assume there are no name clashes between $\delta_1$ and $\delta_2$.)
C consists entirely of well-formedness constraints:

\[ \delta_7 \rightarrow \delta_1 \land \delta_7 \rightarrow \delta_8 \land \delta_8 \rightarrow \delta_2 \land \delta_8 \rightarrow \delta_9 \land \delta_9 \rightarrow \delta_3 \land \delta_9 \rightarrow \delta_6 \land \delta_6 \rightarrow \delta_4 \land \delta_6 \rightarrow \delta_5 \]

However, this constraint plays no further part in this example. Indeed, well-formedness constraints have no impact in this example, so to simplify the presentation, let us treat \( \eta_0 \) as simply

\[ \forall \bar{\delta}. \text{true} \Rightarrow \delta_1 \mapsto \delta_2 \mapsto \delta_3 \mapsto (\delta_4, \delta_5)^{\delta_6} \]

and consider that the initial binding-time environment \( \Gamma_0 \) contains:

\[ \text{snd} : \forall \bar{\delta}. \delta_2 \rightarrow \delta_4 \Rightarrow (\delta_1, \delta_2)^{\delta_3} \rightarrow \delta_4 \]

\[ \text{ite} : \forall \bar{\delta}. \delta_1 \rightarrow \delta_{10} \land \delta_2 \rightarrow \delta_8 \land \delta_3 \rightarrow \delta_9 \land \delta_4 \rightarrow \delta_{10} \land \delta_5 \rightarrow \delta_8 \land \delta_6 \rightarrow \delta_9 \land \delta_7 \rightarrow \delta_{10} \Rightarrow \delta_1 \mapsto (\delta_2, \delta_3)^{\delta_4} \mapsto (\delta_5, \delta_6)^{\delta_7} \mapsto (\delta_8, \delta_9)^{\delta_{10}} \]

Similarly we shall ignore the variable renaming in the (Var-\( \lambda \)) rule.

Inference now proceeds as follows. The body of the lambda abstraction is analysed in the binding-time environment

\[ \Gamma_1 = \Gamma_0, g : \eta_0, p : \delta_7, x : \delta_8, y : \delta_9 \]

For the then-branch we have

\[ \Gamma_1, (x, y) \vdash \inf (\text{true}, (\delta_8, \delta_9)^{\delta_{10}}) \]

That is, no constraints are contributed by this subexpression. For the else-branch, consider first the sub-expression \( g \ p \ y \ x \). Three applications of rule (App) effectively introduce a binding-time type \((\delta_{11}, \delta_{12})^{\delta_{13}}\) for the sub-expression, together with the constraint

\[ (\delta_{14} \mapsto \delta_{15} \mapsto \delta_{16} \mapsto (\delta_{17}, \delta_{18})^{\delta_{19}} \leq_s \delta_7 \mapsto \delta_9 \mapsto \delta_8 \mapsto (\delta_{11}, \delta_{12})^{\delta_{13}}) \]

Notice how a fresh binding-time type \( \delta_{14} \mapsto \delta_{15} \mapsto \delta_{16} \mapsto (\delta_{17}, \delta_{18})^{\delta_{19}} \) has been introduced via rule (Var-Inst).

The constraint translates to

\[ \delta_7 \rightarrow \delta_{14} \land \delta_9 \rightarrow \delta_{15} \land \delta_8 \rightarrow \delta_{16} \land \delta_{17} \rightarrow \delta_{11} \land \delta_{18} \rightarrow \delta_{12} \land \delta_{19} \rightarrow \delta_{13} \]

However, since variables \( \delta_{14} \) to \( \delta_{19} \) are of no further interest (they are neither free in the environment, nor in the result type \((\delta_{11}, \delta_{12})^{\delta_{13}}\)), we can existentially quantify (rule \( \exists \text{Intro} \)):

\[ \exists \delta_{14} \cdots \delta_{19}. \delta_7 \rightarrow \delta_{14} \land \delta_9 \rightarrow \delta_{15} \land \delta_8 \rightarrow \delta_{16} \land \delta_{17} \rightarrow \delta_{11} \land \delta_{18} \rightarrow \delta_{12} \land \delta_{19} \rightarrow \delta_{13} \]

which is equivalent to \( \text{true} \). Hence we have

\[ \Gamma_1, g \ p \ y \ x \vdash \inf (\text{true}, (\delta_{11}, \delta_{12})^{\delta_{13}}) \]
Similarly, for \( \text{snd} \ (g \ p \ y \ x) \) we introduce the result type \( \delta_{20} \), and generate the constraint

\[
\delta_{22} \rightarrow \delta_{24} \land ((\delta_{21}, \delta_{22})^{\delta_{23} \mapsto \delta_{24}} \leq_{s} (\delta_{11}, \delta_{12})^{\delta_{13} \mapsto \delta_{20}})
\]

Again, once we have translated to a Boolean constraint and eliminated uninteresting variables, we are left with the vacuous constraint \( \text{true} \):

\[
\Gamma_1, \text{snd}(g \ p \ y \ x) \vdash_{\text{inf}} (\text{true}, \delta_{20})
\]

It follows that

\[
\Gamma_1, (\text{snd}(g \ p \ y \ x), y) \vdash_{\text{inf}} (\text{true}, (\delta_{20}, \delta_{9})^{\delta_{25}})
\]

Calling the result of \( g \)'s body \((\delta_{26}, \delta_{27})^{\delta_{28}}\), and introducing a fresh instance of the type for \( \text{ite} \), we get

\[
[(\delta_{29} \mapsto \delta_{30}, \delta_{31})^{\delta_{32} \mapsto \delta_{33}, \delta_{34})^{\delta_{35} \mapsto \delta_{36}, \delta_{37})^{\delta_{38}}}
\leq_{s} \delta_{7} \mapsto (\delta_{8}, \delta_{9})^{\delta_{10} \mapsto (\delta_{20}, \delta_{9})^{\delta_{25} \mapsto (\delta_{26}, \delta_{27})^{\delta_{28}}})]
\]

with the additional constraint

\[
\delta_{29} \rightarrow \delta_{38} \land \delta_{30} \rightarrow \delta_{36} \land \delta_{31} \rightarrow \delta_{37} \land \delta_{32} \rightarrow \delta_{38} \land \delta_{33} \rightarrow \delta_{36} \land \delta_{34} \rightarrow \delta_{37} \land \delta_{35} \rightarrow \delta_{38}
\]

The structural constraint translates to

\[
\delta_{7} \rightarrow \delta_{29} \land \delta_{8} \rightarrow \delta_{30} \land \delta_{9} \rightarrow \delta_{31} \land \delta_{10} \rightarrow \delta_{32} \land \delta_{20} \rightarrow \delta_{33} \land \delta_{9} \rightarrow \delta_{34} \land \delta_{25} \rightarrow \delta_{35} \land \delta_{36} \rightarrow \delta_{26} \land \delta_{37} \rightarrow \delta_{27} \land \delta_{38} \rightarrow \delta_{28}
\]

However, only \( \delta_{7}, \delta_{8}, \delta_{9}, \delta_{26}, \delta_{27}, \) and \( \delta_{28} \) are of interest. Existential quantification over the remaining variables yields

\[
\delta_{7} \rightarrow \delta_{28} \land \delta_{8} \rightarrow \delta_{26} \land \delta_{9} \rightarrow \delta_{27}
\]

For the lambda abstraction we therefore obtain

\[
\Gamma_0, g : \eta_{0}, \lambda p. \lambda x. \lambda y. \text{ite} \ p \ (x, y) \ (\text{snd} \ (g \ p \ y \ x), y) \vdash_{\text{inf}} (\delta_{7} \rightarrow \delta_{28} \land \delta_{8} \rightarrow \delta_{26} \land \delta_{9} \rightarrow \delta_{27}, \delta_{7} \mapsto \delta_{8} \mapsto \delta_{9} \mapsto (\delta_{26}, \delta_{27})^{\delta_{28}})
\]

Generalising, we have a new binding-time type scheme \( \eta_{1} \) for \( g \):

\[
g : \forall \vec{\delta}. \delta_{7} \rightarrow \delta_{28} \land \delta_{8} \rightarrow \delta_{26} \land \delta_{9} \rightarrow \delta_{27} \Rightarrow \delta_{7} \mapsto \delta_{8} \mapsto \delta_{9} \mapsto (\delta_{26}, \delta_{27})^{\delta_{28}}
\]

(ignoring well-formedness constraints).

Since \( \eta_{1} \nless \eta_{0} \), the fixed point calculation needs to continue. The effect of \( \eta_{1} \) is to add new constraints when analysing \( g \ p \ y \ x \). We leave it as an exercise for the reader to show that \( \eta_{1} \) is a fixed point. (Hint: \( \text{snd} \) removes the constraints we did not already have.)
We can state soundness and completeness results for well-typed programs. Soundness means that for every deduction in the inference system, we can find an “equivalent” deduction in the logic. We note that the constraints derived in the inference system are Boolean constraints whereas constraints in the logical system are in BTC. Therefore, equivalence here means that the constraints derived in both deduction systems are (semantically) equivalent after translating the constraints in the logical system into Boolean constraints.

**Theorem 1 (Soundness of Inference).** Let $\Gamma, e :: \sigma \vdash_{\inf} (C, \tau)$. Then $C', \Gamma \vdash (e :: \sigma) : \eta$ for some $C'$ such that $\text{gen}(C, \Gamma, \tau) = (C_o, \eta)$ and $[C'] = C_o$.

Completeness states that every deduction derivable in the logical system is subsumed by a deduction in the inference system.

**Theorem 2 (Completeness of Inference).** Let $C, \Gamma \vdash (e :: \sigma) : \forall \bar{\beta}_1, \bar{\delta}_1. C_1 \Rightarrow \tau_1$. Let $\Gamma, e :: \sigma \vdash_{\inf} (C_2, \tau_2)$. Then $[[C \land C_1]] \models \exists \bar{\beta}_2, \bar{\delta}_2. (C_2 \land [[(\tau_2 \leq_s \tau_1)])$ where $\bar{\beta}_2, \bar{\delta}_2 = \text{fv}(C_2, \tau_2) \setminus \text{fv}(\Gamma)$.

In addition, we can state that inference yields principal types.

**Definition 2.** Given a pair $(\Gamma, e :: \sigma)$ consisting of a binding-time environment $\Gamma$ and a type-annotated expression $(e :: \sigma)$, together with a pair $(C, \eta)$, where $C$ is a binding-time constraint and $\eta$ is a binding-time type. $(C, \eta)$ is a principal type of $(\Gamma, e :: \sigma)$ iff

1. $C, \Gamma \vdash (e :: \sigma) : \eta$
2. whenever $C', \Gamma \vdash (e :: \sigma) : \eta'$, we have $C' \vdash C$ and $C' \vdash \eta \leq \eta'$.

(The ordering on binding-time type schemes was defined at the end of Section 3.)

Note that principality is defined with respect to a given type-annotated expression. Hence, annotating an expression with a more general (underlying) type may result in a more general binding-time type. A principal binding-time type represents a class of types which are semantically equivalent. This is in contrast to [4] where principal types are syntactically unique.

**Corollary 1 (Principal Types).** Let $\eta = \forall \bar{\beta}, \bar{\delta}. C \Rightarrow \tau$. Assume $\text{true}, \emptyset \vdash (e :: \sigma) : \eta$ where $e$ is a closed expression. Let $(\text{true}, \eta)$ be the principal type of $(\emptyset, e :: \sigma)$. Then $\emptyset, e :: \sigma \vdash_{\inf} (C, \tau)$.

## 6 Alternative Methods for Handling Polymorphism

With certain constraint solvers the instantiate method (rule Var-Inst) is impractical. They store the constraints in an optimised internal form; to reverse engineer this representation to find which relations hold between polymorphic variables may be very inefficient, if not impossible.

However, it is always possible for the solver to determine if a constraint set entails a particular relationship amongst variables. Many solvers can do this efficiently. The Test-Add method uses entailment to avoid constraint instantiation.
6.1 The Test-Add Method

Instead of instantiating variables in the constraint component of the type scheme \( \forall \bar{\beta}, \bar{\delta}. C \Rightarrow \tau \), the “Test–Add” method queries which implications hold between elements of \( \bar{\beta} \) in \( C \). For those that hold we add inequalities between the corresponding instantiating type variables to \( C \).

Note we have two forms of constraints involving polymorphic binding-time variables. \( \beta \rightarrow \beta' \) (connects two polymorphic binding-time variables), and \( \delta \rightarrow \beta \) (a well-formedness condition or generated from a conditional expression such as if-then-else).

Consider a binding-time type scheme \( \eta = \forall \bar{\beta}, \bar{\delta}. C \Rightarrow \tau \), an underlying type scheme \( \sigma = \forall \bar{\alpha}. t \), a shape environment \( \Delta \) and a sequence of instantiation types \( \tau_{ij} \) such that \( \Delta \vdash \tau : t \) and for each \( \Delta \vdash \beta_{ij} : \alpha_i \) it holds that \( \Delta \vdash \tau_{ij} : \alpha_i \). We define

\[
T_{\alpha_i}(\Delta, \tau, \bar{\tau}_i) = \begin{cases} 
\Delta \vdash \beta_{ij} : \alpha_i, \tau[\beta_{ij}], & \text{if } \tau_{ij} \leq_s \tau_{ik} \\
\Delta \vdash \beta_{ik} : \alpha_i, \tau[\beta_{ik}], & \text{if } \tau_{ij} \geq_s \tau_{ik}
\end{cases}
\]

\[
T(\Delta, \tau, \bar{\tau}) = \bigwedge_{\alpha_i \in \text{range}(\Delta)} T_{\alpha_i}(\Delta, \tau, \bar{\tau}_i)
\]

Recall that we only consider directed constraints (Definition 1) and the inference rules preserve this condition.

The set of generated constraints is a subclass of HORN. Hence, the test \( C \models (\beta_{ij} \rightarrow \beta_{ik}) \) can be performed in linear time. It remains to handle constraints of the form \( (\delta \leq f \beta) \). In Example 3 we had the constraint \( (\delta_2 \leq f \beta_1) \) where \( \delta_2 \) was instantiated to \( \delta_{13} \) and \( \beta_1 \) to \( \delta_{10} \rightarrow \delta_{12} \). We define

\[
A_{\alpha_i}(\bar{\beta}_i, \bar{\tau}_i) = \bigwedge \{ \beta_{ij} \rightarrow [\tau_{ij}] \}
\]

\[
A(\Delta, \bar{\beta}, \bar{\tau}) = \bigwedge_{\alpha_i \in \text{range}(\Delta)} A_{\alpha_i}(\bar{\beta}_i, \bar{\tau}_i)
\]

where \(|\tau|\) refers to \( \tau \)’s top-level annotation and is defined as follows:

\[
|b| = b \quad |\beta| = \beta \quad |\tau \xrightarrow{b} \tau'| = b
\]

This ensures that if we have a relation \( \delta \rightarrow \beta_{ij} \) in the constraint component of the type scheme, then together with \( A_{\alpha_i} \) we obtain \( \delta \rightarrow |\tau_{ij}| \).

Example 6. We reconsider Example 3. Since \( \beta_1 \rightarrow \beta_3 \) holds in \( \text{id} \)’s binding-time type, the \( T \) operator yields the constraint \( [(\delta_{10} \rightarrow \delta_{12}) \leq_s \delta_{14} \rightarrow \delta_{15}] \). The \( A \) operator yields the constraint \( \beta_1 \rightarrow \delta_{11} \land \beta_3 \rightarrow \delta_{15} \). Adding \( \text{id} \)’s constraint component \( [(\beta_1 \leq_s \beta_3) \land (\delta_2 \leq f \beta_1) \land (\delta_2 \leq f \beta_3)] \), we obtain exactly the same result as the instantiation method.

The following states that “Instantiate” and “Test–Add” are equivalent.
Lemma 4. Given a binding-time type scheme $\eta = \forall \bar{\beta}, \bar{\delta}. C \Rightarrow \tau$, an underlying type scheme $\sigma = \forall \bar{\alpha}. t$, a shape environment $\Delta$ and a sequence of instantiation types $\tau_{ij}$ such that $\Delta \vdash \tau : t$ and for each $\Delta \vdash \beta_{ij} : \alpha_i$ it holds that $\Delta \vdash \tau_{ij} : \alpha_i$.

\[
[[\bar{\tau} / \bar{\beta}] C] \equiv \exists \bar{\beta}. ([[[C] \land T(\Delta, \tau, \bar{\tau}) \land A(\Delta, \bar{\beta}, \bar{\tau})].
\]

Lemma 4 allows us to replace rule (Var–Inst) with the equivalent rule (Var–Inst’):

$$(\text{Var–Inst'}) \quad x : \forall \bar{\beta}, \bar{\delta}. C \Rightarrow \tau \in \Gamma \quad t' = [\bar{t} / \bar{\alpha}] t
\tau, t \vdash \Delta \quad \text{inst}(\Delta, \bar{t}, \bar{\alpha}) = \bar{\tau}
(\text{Var–Inst'}) \quad C' = [[\bar{\delta}' / \bar{\delta}] C] \land T(\Delta, \tau, \bar{\tau}) \land A(\Delta, \bar{\beta}, \bar{\tau})
\tau' = [\bar{\tau} / \bar{\beta}, \bar{\delta}' / \bar{\delta}] \tau \quad \text{\ new } \bar{\delta}'
\Gamma, ((x :: \forall \bar{\alpha}. t) :: t') \vdash_{\inf} (C', \tau')$$

Theorem 3 (Polymorphic Application 1). Let $\Gamma$ be directed. Assume $\Gamma, e :: t \vdash_{\inf} (C, \tau)$ using rule (Var–Inst) while $\Gamma, e :: t \vdash_{\inf} (C', \tau')$ using rule (Var–Inst'). Then $C \equiv \pi(C')$ and $\tau = \pi(\tau')$ for some renaming $\pi$.

6.2 The Match Method

We briefly comment on further alternative methods. For more details we refer to [10]. In “Test–Add” we query possible relations between binding-time variables which are related by the shape environment to the same underlying type variable. We may wish to avoid this querying during the actual analysis. The “Match” method allows us to simply add constraints which will give a similar effect.

The method works by matching the type component of the variable’s type scheme with the requested type at the instantiation site.

Example 7. Consider again Example 3. First generate a new binding-time type of the desired form: $(\delta_{10} \overset{\delta_{11}}{\rightarrow} \delta_{12}) \overset{\delta_{13}}{\rightarrow} \delta_{14} \overset{\delta_{15}}{\rightarrow} \delta_{16}$.

Instead of substituting the appropriate types for $\beta_1, \beta_3$ and $\delta_2$, generate constraints as follows:

$$(\beta_1 \overset{\delta_2}{\rightarrow} \beta_3 \leq_s (\delta_{10} \overset{\delta_{11}}{\rightarrow} \delta_{12}) \overset{\delta_{13}}{\rightarrow} (\delta_{14} \overset{\delta_{15}}{\rightarrow} \delta_{16}))$$

Note that the polymorphic binding-time variables $\beta_1$ and $\beta_3$ result from the polymorphic variable $\alpha$. How do we translate the structural constraints into Boolean constraints? Since in this case $\text{id}$’s constraints support $\beta_1 \rightarrow \beta_3$ we should have that $(\delta_{10} \overset{\delta_{11}}{\rightarrow} \delta_{12} \leq_s \delta_{14} \overset{\delta_{15}}{\rightarrow} \delta_{16})$. If $\text{id}$ did not support $\beta_1 \rightarrow \beta_3$ then we should not have this constraint. This reasoning is captured by the constraint:

$$(\beta_1 \rightarrow \beta_3) \rightarrow [[((\delta_{10} \overset{\delta_{11}}{\rightarrow} \delta_{12} \leq_s \delta_{14} \overset{\delta_{15}}{\rightarrow} \delta_{16}))]]$$

$$(\beta_1 \rightarrow \beta_3) \rightarrow (\delta_{14} \rightarrow \delta_{10} \land \delta_{11} \rightarrow \delta_{15} \land \delta_{12} \rightarrow \delta_{16})$$
7 Constraint Based Fixed Points

If we are willing to give up some accuracy in the presence of polymorphic recursion in binding-time types then we can replace the (Fix) rule by a simpler rule which just ensures that the constraints generated will be above the least fixed point, while avoiding a fixed point computation.

The following rule forces all recursive invocations to have the same binding-time type. This must be a super-type of the binding-time type of the recursively defined expression.

\[
\begin{align*}
\Delta, t \vdash \tau' &\quad \Gamma_x.x : \tau', e :: t \vdash \inf (C_1, \tau) \\
(\text{FixC}) &\quad \Delta, t \vdash \tau'' \\
C = C_1 \land [\lfloor (\tau \leq_s \tau') \rfloor] \land [\lfloor (\tau \leq_s \tau'') \rfloor] \land [\lfloor \text{wft}(\tau') \rfloor] \land [\lfloor \text{wft}(\tau'') \rfloor] \\
\Gamma_x.(\text{fix } x :: t \in e) :: t \vdash \inf (C, \tau'')
\end{align*}
\]

The “shortcut” of stipulating \((\tau \leq_s \tau')\) may have the unfortunate side-effect of introducing constraints amongst the arguments to a function. An example of this phenomenon appears in Example 8 below. A simple way of eliminating such constraints is to couch the result in terms of a fresh binding-time type \(\tau''\), with the constraint \((\tau \leq_s \tau'')\).

The following shows how this approach may lose accuracy, finding a correct but inaccurate binding-time type for a function.

Example 8. Consider again \(g\) from Example 5:

\[
g : \text{Bool} \to \text{Int} \to \text{Int} \to (\text{Int}, \text{Int})
\]

\[
g p x y = \text{if } p \text{ then } (x,y) \text{ else } (\text{snd} (g p y x), y)
\]

The binding-time type inferred using the rule (Fix) was

\[
g : \forall \delta.\delta_7 \to \delta_{28} \land \delta_8 \to \delta_{26} \land \delta_9 \to \delta_{27} \Rightarrow \delta_7 \mapsto \delta_8 \mapsto \delta_9 \mapsto (\delta_{26}, \delta_{27})^{\delta_{28}}
\]

This time, the initial environment \(\Gamma\) has

\[
g : \delta_1 \mapsto \delta_2 \mapsto \delta_3 \mapsto (\delta_4, \delta_5)^{\delta_6}
\]

so that \(\delta_1 \ldots \delta_6\) are not generic. For the subexpression \(g p y x\), we now get

\[
(\delta_1 \mapsto \delta_2 \mapsto \delta_3 \mapsto (\delta_4, \delta_5)^{\delta_6} \leq_s \delta_7 \mapsto \delta_9 \mapsto \delta_8 \mapsto (\delta_{11}, \delta_{12})^{\delta_{13}})
\]

This translates to

\[
\delta_7 \to \delta_1 \land \delta_9 \to \delta_2 \land \delta_8 \to \delta_3 \land \delta_4 \to \delta_{11} \land \delta_5 \to \delta_{12} \land \delta_6 \to \delta_{13}
\]

and none of the variables involved can be discarded. For \(\text{snd} (g p y x)\) we introduce the result type \(\delta_{14}\), and add the constraint

\[
\delta_{16} \to \delta_{18} \land [((\delta_{15}, \delta_{16})^{\delta_{17}} \mapsto \delta_{18} \leq_s (\delta_{11}, \delta_{12})^{\delta_{13}} \mapsto \delta_{14})]
\]
that is,
\[ \delta_{16} \rightarrow \delta_{18} \land \delta_{11} \rightarrow \delta_{15} \land \delta_{12} \rightarrow \delta_{16} \land \delta_{13} \rightarrow \delta_{17} \land \delta_{18} \rightarrow \delta_{14} \]

Here variables \( \delta_{11} \ldots \delta_{13} \) and \( \delta_{15} \ldots \delta_{18} \) can be discarded. Existentially quantifying over these leaves us with
\[ \delta_7 \rightarrow \delta_1 \land \delta_9 \rightarrow \delta_2 \land \delta_8 \rightarrow \delta_3 \land \delta_5 \rightarrow \delta_{14} \]
So
\[ \Gamma, \text{snd}(g \ p \ y \ x) \models_{inf} (\delta_7 \rightarrow \delta_1 \land \delta_9 \rightarrow \delta_2 \land \delta_8 \rightarrow \delta_3 \land \delta_5 \rightarrow \delta_{14}, \ \delta_{14}) \]

It follows that
\[ \Gamma, (\text{snd}(g \ p \ y \ x), y) \models_{inf} (\delta_7 \rightarrow \delta_1 \land \delta_9 \rightarrow \delta_2 \land \delta_8 \rightarrow \delta_3 \land \delta_5 \rightarrow \delta_{14}, \ (\delta_{14}, \delta_9)_{\delta_{19}}) \]

Calling the result of \( g \)’s body \( (\delta_{20}, \delta_{21})_{\delta_{22}} \), we get for the expression
\[ \left[ (\delta_{23} \rightarrow (\delta_{24}, \delta_{25})_{\delta_{26}} \rightarrow (\delta_{27}, \delta_{28})_{\delta_{29}} \rightarrow (\delta_{30}, \delta_{31})_{\delta_{32}} \leq_s \delta_7 \rightarrow (\delta_8, \delta_9)_{\delta_{10}} \rightarrow (\delta_{14}, \delta_9)_{\delta_{19}} \rightarrow (\delta_{20}, \delta_{21})_{\delta_{22}} \right] \]
with the additional constraint
\[ \delta_{23} \rightarrow \delta_{32} \land \delta_{24} \rightarrow \delta_{30} \land \delta_{25} \rightarrow \delta_{31} \land \delta_{26} \rightarrow \delta_{32} \land \delta_{27} \rightarrow \delta_{30} \land \delta_{28} \rightarrow \delta_{31} \land \delta_{29} \rightarrow \delta_{32} \]
The structural constraint translates to
\[ \delta_7 \rightarrow \delta_{23} \land \delta_8 \rightarrow \delta_{24} \land \delta_9 \rightarrow \delta_{25} \land \delta_{10} \rightarrow \delta_{26} \land \delta_{14} \rightarrow \delta_{27} \land \delta_9 \rightarrow \delta_{28} \land \delta_{19} \rightarrow \delta_{29} \land \delta_{30} \rightarrow \delta_{20} \land \delta_{31} \rightarrow \delta_{21} \land \delta_{32} \rightarrow \delta_{22} \]
Variables of interest are \( \delta_1 \ldots \delta_9 \) and \( \delta_{20} \ldots \delta_{22} \). Eliminating the rest, we get
\[ \delta_7 \rightarrow \delta_1 \land \delta_9 \rightarrow \delta_2 \land \delta_8 \rightarrow \delta_3 \land \delta_5 \rightarrow \delta_{20} \land \delta_7 \rightarrow \delta_{22} \land \delta_8 \rightarrow \delta_{20} \land \delta_9 \rightarrow \delta_{21} \]
For the lambda abstraction we therefore obtain the result
\[ (\delta_7 \rightarrow \delta_1 \land \delta_9 \rightarrow \delta_2 \land \delta_8 \rightarrow \delta_3 \land \delta_5 \rightarrow \delta_{20} \land \delta_7 \rightarrow \delta_{22} \land \delta_8 \rightarrow \delta_{20} \land \delta_9 \rightarrow \delta_{21}, \quad \delta_7 \rightarrow \delta_8 \rightarrow \delta_9 \mapsto (\delta_{20}, \delta_{21})_{\delta_{22}}) \]
At this point, the rule (FixC) adds the following two inequalities
\[ \left[ (\delta_7 \rightarrow \delta_8 \rightarrow \delta_9 \mapsto (\delta_{20}, \delta_{21})_{\delta_{22}} \leq_s \delta_{11} \mapsto \delta_2 \mapsto \delta_3 \mapsto (\delta, \delta_5)_{\delta_{9}} \right] \]
and
\[ \left[ (\delta_7 \rightarrow \delta_8 \rightarrow \delta_9 \mapsto (\delta_{20}, \delta_{21})_{\delta_{22}} \leq_s \delta_{33} \mapsto \delta_{34} \mapsto \delta_{35} \mapsto (\delta_{36}, \delta_{37})_{\delta_{38}} \right] \]
Altogether we then have
\[ \delta_7 \rightarrow \delta_1 \land \delta_9 \rightarrow \delta_2 \land \delta_8 \rightarrow \delta_3 \land \delta_5 \rightarrow \delta_{20} \land \delta_7 \rightarrow \delta_{22} \land \delta_8 \rightarrow \delta_{20} \land \delta_9 \rightarrow \delta_{21} \land \delta_1 \rightarrow \delta_7 \land \delta_2 \rightarrow \delta_8 \land \delta_3 \rightarrow \delta_9 \land \delta_{20} \rightarrow \delta_4 \land \delta_{21} \rightarrow \delta_5 \land \delta_{22} \rightarrow \delta_6 \land \delta_{33} \rightarrow \delta_7 \land \delta_{34} \rightarrow \delta_8 \land \delta_{35} \rightarrow \delta_9 \land \delta_{20} \rightarrow \delta_{36} \land \delta_{21} \rightarrow \delta_{37} \land \delta_{22} \rightarrow \delta_{38} \]
Note that this has spurious consequences such as $\delta_8 \leftrightarrow \delta_9$. However, for the result, we are only interested in the freshly introduced $\delta_{33} \cdots \delta_{38}$, so spurious consequences are removed. After existential quantification we obtain the (weaker) result (changes relative to Example 5 are underlined):

$$g : \forall \delta. \delta_{33} \rightarrow \delta_{38} \land \delta_{34} \rightarrow \delta_{36} \land \delta_{35} \rightarrow \delta_{37} \land \delta_{34} \rightarrow \delta_{37} \land \delta_{35} \rightarrow \delta_{36} \Rightarrow \delta_{33} \mapsto \delta_{34} \mapsto \delta_{35} \mapsto (\delta_{36}, \delta_{37})^{\delta_{38}}$$

**Theorem 4.** The FixC rule is sound.

8 Discussion

We have presented a binding-time analysis with a wide scope. The analysis is polyvariant and extends Dussart, Henglein and Mossin’s analysis to polymorphically typed programs. It applies to a functional programming language with ML-style polymorphism. The handling of (type) polymorphic application is not straightforward. We have outlined some options, utilising the increased expressiveness that we obtain by using Boolean constraints.

Types provide useful information about a program’s properties and can readily be extended for various program analyses for higher-order functional languages. A recent trend is to present (and implement) program analysis as “non-standard type inference”, by marrying a type language with notation for decorating type expressions, the decorations expressing the program properties of interest. The direction that much of this research is currently taking is to extend the underlying type language beyond what the programming language requires, for example to include intersection types or subtyping.

Our view is that, while it is both convenient and elegant to express program analysis as constrained-type inference, the language for expressing program properties should not necessarily be coupled closely with the type system. From the point of view of the analysis designer and implementor, it seems more attractive to utilise expressive constraint languages that come with well-developed solvers and well-understood theories, making only the reasonable assumption that programs presented to the analyser are well-typed and explicitly typed, for example by an earlier type inference phase in a compiler.

We believe that propositional logic is under-utilised in the analysis of functional programs. It offers straightforward means for expressing dependencies and disjunctive information. Indeed, some of our proposed solutions to the problem of analysis in the presence of polymorphic application are expressed through the use of non-trivial propositional formulas such as $(\beta \rightarrow \beta') \rightarrow C$, as seen in Example 7.

We have a prototype implementation of the binding-time analysis presented here for the Haskell compiler, GHC. Our implementation takes a Haskell program

---

2 Dussart, Henglein and Mossin have disjunction in their language of binding-time properties, but only as a technical aid for the elimination of variables (as they do not have variable projection).
and assigns a binding-time type to every sub-expression. The information could be used by a program specialiser, for example Henglein and Mossin’s suitably extended to support type polymorphic programs. However, we have no such specialiser, and our interest is primarily the analysis of functional languages.

The implementation includes necessary extensions to the analysis described here so that the language accepted by GHC is supported. In particular, it supports algebraic data types, and corresponding constructors and case statements.

GHC translates the input program into a de-sugared, typed internal language, Core. A GHC analysis or optimisation pass is a plug-in module that transforms an input Core program into a transformed Core program (maintaining type correctness). Since type information is explicit in the Core program the implementation is a direct translation of the analysis here. Our BTA pass annotates variables with their BTA properties. An increasing number of modern compilers have a similar internal language and it should not be difficult to incorporate our implementation within these.

We have experimented with two constraint solvers, Schachte’s ROBDD library for Boolean constraints and ImpSolver, a straightforward solver that is limited to conjunctions of implications between variables. ImpSolver is sufficient for this analysis if we employ the Test-Add method for polymorphic application. A third solver, for propositional Horn clauses, is under development for other functional program analyses.

We have run the ImpSolver based analyser on the NoFib suite of benchmark programs which is available with GHC. This suite consists of over 60 (multi-module) programs. For the 68 larger modules, those that take over 2 seconds to compile on our lightly loaded server, the average cost of the analysis is 23% of compile time, with 16% median, 1% minimum and 80% maximum. This is before non-trivial performance tuning has been attempted.

The implementation is at an early stage of evaluation. It does not yet support cross-module analysis (instead we make a ‘good’ guess for imported variables). Adding this is straightforward: with each exported binding in the module’s interface file we add its binding-time type and list the implications which hold amongst the binding-time variables. When a binding is imported, we recreate the corresponding constraints and add the binding to the initial environment.

We believe that the ideas presented here have many immediate applications. Binding-time analysis is essentially identifying dependencies amongst components of expressions, and this is also the heart of several other analyses. For example, this is the case for security analysis and for useless-code analysis, program slice analysis etc. Several researchers have addressed the problem of how to systematically extend a type system with dependency information. The goal of these researchers is similar to ours, but generally more limited with respect to scope.

Fähndrich and Rehof have recently proposed a new method for flow analysis. In the absence of recursive types, the method offers improved worst-case time complexity for flow analysis, namely cubic time analysis. The key to the improvement is a concept of “instantiation” constraints. This is an extension of
the constraint language which offers a way of avoiding the copying of constraints that otherwise happens when a polymorphic type is instantiated. The constraint solving problem is then translated to one of CFL reachability. A different kind of extension of the constraint language is proposed by Gustavsson and Svenningsson [11]. A “let” construct allows for “constraint abstractions”, so that instantiation can be expressed directly in the constraint language. Again this leads to a cubic-time algorithm. The authors point out that the idea of incorporating instantiation into a constraint language goes back at least to Henglein’s use of semi-unification constraints [13]. We are currently investigating these methods and hope to explore whether, for our implementations, the improved complexity will translate into faster analysis.

We are now developing a generic framework for specifying and implementing program analyses, based on constrained-type inference. We see this binding-time analysis as a first instance, that is, a proof of concept. We are currently applying the framework to strictness and flow analysis.

References

Abstract. Many type based program analyses with subtyping, such as flow analysis, are based on inequality constraints over a lattice. When inequality constraints are combined with polymorphism it is often hard to scale the analysis up to large programs. A major source of inefficiency in conventional implementations stems from computing substitution instances of constraints. In this paper we extend the constraint language with constraint abstractions so that instantiation can be expressed directly in the constraint language and we give a cubic-time algorithm for constraint solving. As an application, we illustrate how a flow analysis with flow subtyping, flow polymorphism and flow-polymorphic recursion can be implemented in $O(n^3)$ time where $n$ is the size of the explicitly typed program.

1 Introduction

Constraints are at the heart of many modern program analyses. These analyses are often implemented by two stages. The first stage collects constraints in an appropriate constraint language and the second stage finds a solution (usually the least) to the constraints. If the constraints are collected through a simple linear time traversal over the program yielding a linear amount of constraints the first phase can hardly constitute a bottleneck. But often the constraints for a program point are computed by performing a non constant-time operation on constraints collected for another part of the program. Notable examples, and the motivation for this work, are analyses which combine subtyping and polymorphism. There, typically, the constraints for a call to a polymorphic function $f$ are a substitution instance of the constraints for the body of $f$. For these analyses, to naively collect constraints typically leads to unacceptable performance. Consider, for example, how we naively could collect the constraints for a program of the following form.

```plaintext
let $f_0 = \ldots$
in let $f_1 = \ldots f_0 f_0$
in let $\ldots$
in let $f_n = \ldots f_{n-1} f_{n-1} \ldots$
in $\ldots f_n f_n \ldots$
```

We first collect the constraints for the polymorphic function $f_0$. Then for the two calls to $f_0$ in the body of $f_1$, we compute two different substitution instances of the constraints from the body of $f_0$. As a result the number of constraints for
\( f_1 \) will be at least twice as many as those for \( f_0 \). Thus, the number of resulting constraints grows exponentially in the call depth \( n \) (even if the underlying types are small). In analyses which combine subtyping and polymorphic recursion, and rely on a fixed point iteration, this effect may show up in every step of the iteration and thus the constraints may grow exponentially in the number of required iterations. We can drastically reduce the number of constraints if we can simplify the constraints to fewer but equivalent constraints. It is therefore no surprise that lots of work has been put into techniques for how to simplify constraints [FM89, Cur90, Kae92, Smi94, EST95, Pot96, TS96, FA96, AWP97, Reh97, FF97].

Another approach is to make the constraint language more powerful so that constraints can be generated by a simple linear time traversal over the program. This can be achieved by making substitution instantiation a syntactic construct in the constraint language. But when we make the constraint language more powerful we also make constraint solving more difficult. So is this a tractable approach? The constraint solver could of course just perform the delayed operations and then proceed as before. But can one do better? The answer, of course, depends on the constraint language in question.

In this paper we consider a constraint language with simple inequality constraints over a lattice. Such constraints show up in several type based program analyses such as flow analyses, e.g., [Mos97], binding time analyses, e.g., [DHM95], usage analyses, e.g., [JAW95], points-to-analyses, e.g., [FFAU00] and uniqueness type systems [BS96]. We extend this simple constraint language with constraint abstractions which allow the constraints to compactly express substitution instantiation.

The main result of this paper is a constraint solving algorithm which computes least solutions to the extended form of constraints in cubic time. We have used this expressive constraint language to formulate usage-polymorphic usage analyses with usage subtyping [Sve00, GS00] and an algorithm closely related to the one in this paper is presented in the second author’s Master’s thesis [Sve00] (GS00 focuses on the usage type system and no constraint solving is presented). In this paper, as another example, we show how the constraint language can be used to yield a cubic algorithm for Mossin’s polymorphic flow analysis with flow subtyping and flow-polymorphic recursion [Mos97]. This is a significant result – the previously published algorithm, by Mossin, is \( O(n^8) \). Independently, Fähndrich and Rehof [RF01] have given an algorithm for Mossin’s flow analysis based on instantiation constraints which is also \( O(n^3) \). We will take a closer look at the relationship of their algorithm and ours in section 4.

1.1 Outline

The rest of this article is organised as follows. In section 2 we introduce our constraint language and give the semantics. In section 3 we present our constraint solving algorithm, its implementation and computational complexity. Section 4 discusses related work and section 5 concludes. In appendix A we illustrate how
the constraint language can be used in a flow analysis. In appendix B we give the proof of Theorem 1.

2 Constraints

In this section we will first introduce the underlying constraints language that we consider in this paper, and then extend the constraint language with constraint abstractions which can express substitution instantiation. The atomic constraints we consider are inequality constraints of the form

\[ a \leq b \]

where \( a \) and \( b \) are taken from a countably infinite set of variables. The constraint language also contains the trivially true constraint, conjunction of constraints and existential quantification as given by the following grammar.

**Atomic Constraints**

\[ A ::= a \leq b \]

**Constraint Terms**

\[ M, N ::= A \mid \top \mid M \land N \mid \exists a. M \]

These kinds of constraints show up in several different type based program analyses such as, for example, flow analysis, e.g., [Mos97] which we will use as our running example. The constraints arise from the use of subtyping between flow types - i.e., types annotated with flow information.

Depending on the application, the constraints can be interpreted in different domains. For example, for flow analysis we can interpret the constraints in a lattice of finite sets of labels with subset as the ordering.

**Definition 1.** We interpret a constraint term in a lattice \( \mathcal{L} \), with a bottom element and the ordering \( \subseteq \), by defining the notion of a model of a constraint term. Let \( \theta \) range over mappings from variables into \( \mathcal{L} \). Then \( \theta \models M \), read as \( \theta \) is a model of \( M \), is defined inductively by the following rules.

\[
\theta(a) \subseteq \theta(b) \quad \frac{\theta \models a \leq b}{\theta \models \top} \quad \frac{\theta \models M \quad \theta \models N}{\theta \models M \land N} \quad \frac{\theta \models \exists a. M}{\theta[d := d] \models M} \\
\text{where } d \in \mathcal{L}
\]

Given a constraint term one is usually interested in finding its optimal model (usually the least) given a fixed assignment of some of the variables. For example, in flow analysis some of the variables in the constraint term correspond to points in the program where values are produced, often referred to as the sources of flow. Other variables correspond to points in the program where values are consumed, often referred to as the targets of flow. The existentially quantified variables correspond to the flow annotations on intermediate flow types. To find the flow from the sources to the targets we can fix an assignment for the source variables (usually by associating a unique label \( l \) to each source and interpret it as the singleton set \( \{l\} \)) and compute the least model which respects this assignment. For this simple constraint language it is easy to compute least solutions (it can
be seen as a transitive closure problem) in $O(n^3)$ time, where $n$ is the number of variables.\footnote{For a lattice where binary least upper bounds can be computed in constant time (for example a two point lattice) the least solution can be computed in $O(n^2)$ time.}

## 2.1 Constraint Abstractions

When subtyping is combined with polymorphism the need to compute substitution instances of constraint terms arise. We will build this operation into our constraint language through the means of constraint abstractions.

Constraint Abstraction Variables $f, g, h$
Constraint Abstractions $F ::= f \ a = M$

A constraint abstraction $f \ a = M$ can be seen simply as a function which when applied to some variables $b$ returns $M[a := b]$. Constraint abstractions are introduced by a let-construct reminiscent of let-con structs in functional languages, and are also called in the same way. The complete grammar of the extended constraint language is as follows.

Atomic Constraints $A ::= a \leq b$
Constraint Terms $M, N ::= A | \top | M \land N | \exists a. M | \text{let } \{F\} \text{ in } M | f \ a$
Constraint Abstractions $F ::= f \ a = M$

We will write $\text{FV}(M)$ for the free variables of $M$ and $\text{FAV}(M)$ for the free abstraction variables of $M$. We will identify constraint terms up to $\alpha$-equivalence, that is the renaming of bound variables and bound abstraction variables. In $\text{let } \{F\} \text{ in } M$ the constraint abstraction variables defined by $F$ are bound both in $M$ and in the bodies of $F$ so our lets are mutually recursive. Consequently the variables defined by $F$ must be distinct. We will use $\Gamma$ to range over sets of constraint abstractions where the defined variables are distinct, and we will denote the addition of a group of distinct constraint abstractions $F$ to $\Gamma$ by juxtaposition: $\Gamma \{F\}$. We will say that a group of constraint abstractions $F$ is garbage in $\text{let } \Gamma \{F\} \text{ in } M$ if we can remove the abstractions without causing bound abstraction variables to become free. Recursive constraint abstractions goes beyond just expressing a delayed substitution instantiation. It also allows us to express a fixed-point calculation in a very convenient way. We will make use of this in the flow analysis in appendix A to express flow-polymorphic recursion.

To give a semantics to the extended constraint language we need to define the notion of a model of a constraint term in the context of a set of constraint abstractions $\Gamma$.

**Definition 2.** In a lattice $\mathcal{L}$, with a bottom element and with the ordering $\sqsubseteq$, we define $\theta; \Gamma \models M$ coinductively by the following rules (we follow the notational convention of Cousot and Cousot \cite{CC92} to mark the rules with a “$-$” to indicate that it is a coinductive definition).
The definition needs to be coinductive to cope with recursive constraint abstractions. The coinductive definition expresses the intuitive concept that such constraint abstractions should be “unfolded infinitely”. When it is not clear from the context we will write $\theta; \Gamma \models L M$ to make explicit which lattice we consider. We will say that $N$ is a consequence of $M$, written $M \models N$, iff for every $L, \theta, \Gamma$, if $\theta; \Gamma \models L M$ then $\theta; \Gamma \models L N$. We will write $M \Leftrightarrow N$ iff $M \models N$ and $N \models M$.

In definitions throughout this paper we will find it convenient to work with constraint term contexts. A constraint term context is simply a constraint term with a “hole” analogous to term contexts used extensively in operational semantics.

**Constraint Term Contexts**

$$C ::= [\cdot] \mid C \land M \mid M \land C \mid \exists a.C \mid \text{let } \Gamma \text{ in } C \mid \text{let } \Gamma \{f a = C\} \text{ in } M$$

We will write $C[M]$ to denote the filling of the hole in $C$ with $M$. Hole filling may capture variables. We will write $CV(C)$ for the variables that may be captured when filling the hole. We will say that the hole in $C$ is live if the hole does not occur in a constraint abstraction which is garbage. Our first use of constraint term contexts is in the definition of the free live atomic constraints of a constraint term.

**Definition 3.** The set of free live atomic constraints of a constraint term $M$, denoted $LIVE(M)$, is defined as follows.

$$LIVE(M) = \{ A \mid M \equiv C[A], FV(A) \cap CV(C) = \emptyset \text{ and the hole in } C \text{ is live.} \}$$

We will use $LIVE(M)$ in definitions where we need to refer to the atomic subterms of $M$ but want to exclude those which occur in constraint abstractions which are garbage and thus never will be “called” by the models relation. Note that all syntactically live constraint abstractions are semantically live since they are all “called” by the models relation.

Another use of constraint term contexts is in the statement of the following unwinding lemma.

**Lemma 1.** If $FV(M) \cap CV(C) = \emptyset$ then

$$\text{let } \Gamma \{f a = M\} \text{ in } C[f b] \Leftrightarrow \text{let } \Gamma \{f a = M\} \text{ in } C[M[a := b]]$$

This lemma is necessary, and is the only difficulty, when proving the subject reduction property of the usage analysis in [GS00] and the flow analysis in appendix.
1. if $a \leq b, b \leq c \in \text{LIVE}(M)$ then
   \[ \exists b. M \mapsto \exists b. M \land a \leq c \]

2. if $A \in \text{LIVE}(M)$, and, for some $i, a_i \in \text{FV}(A)$ then
   \[
   \begin{align*}
   \text{let } & \Gamma \{ f a = M \} \mapsto \text{let } \Gamma \{ f a = M \} \\
   \text{in } & C[f b] \quad \text{in } C[f b \land A[a := b]]
   \end{align*}
   \]

3. if $A \in \text{LIVE}(C[f b])$, and, for some $i, a_i \in \text{FV}(A)$ then
   \[
   \begin{align*}
   \text{let } & \Gamma \{ f a = C[f b] \} \mapsto \text{let } \Gamma \{ f a = C[f b \land A[a := b]] \} \\
   \text{in } & M \quad \text{in } M
   \end{align*}
   \]

4. if $A \in \text{LIVE}(M)$, and for some $i, a_i \in \text{FV}(A)$ then
   \[
   \begin{align*}
   \text{let } & \Gamma \{ f a = M \} \{ g c = C[f b] \} \mapsto \text{let } \Gamma \{ f a = M \} \{ g c = C[f b \land A[a := b]] \} \\
   \text{in } & M \quad \text{in } M
   \end{align*}
   \]

Fig. 1. Rewrite rules

The premise $\text{FV}(M) \cap \text{CV}(C) = \emptyset$ is there to ensure that no inadvertent name capture takes place and it can always be fulfilled by an $\alpha$-conversion. In the remainder of this paper we will leave this condition on unwindings implicit.

3 Solving Constraints

As we discussed in the previous section we are interested in finding the least model of a constraint term given a fixed assignment of some of the variables. In this section we will present an algorithm for this purpose for our constraint language. The algorithm is based on a rewrite system which rewrites constraint terms to equivalent but more informative ones. Every rewrite step adds an atomic constraint to the constraint term and the idea is that when the rules have been applied exhaustively then enough information is explicit in the term so that the models can be constructed easily.

**Definition 4.** We define the rewrite relation $\rightarrow$ as the compatible closure of the relation $\rightarrow$ defined by the clauses in figure 1.

Here we provide some explanation of the rewrite rules. The first rule,

1. if $a \leq b, b \leq c \in \text{LIVE}(M)$ then
   \[ \exists b. M \mapsto \exists b. M \land a \leq c \]

is a simple transitivity rule. If $a \leq b$ and $b \leq c$ are free live atomic subterms of $M$ we may simply add the constraint $a \leq c$. Note that the rule requires $a$ and $c$
to be in scope at the binding occurrence of $b$. As a result we cannot, for example, perform the rewrite

$$\exists a. \exists b. (a \leq b) \land (\exists c. b \leq c) \rightarrow \exists a. \exists b. (a \leq b) \land (\exists c. b \leq c \land a \leq c)$$

which adds $a \leq c$ although it would make perfect sense. The reason is simply that at the binding occurrence of $b$, $c$ is not in scope. The purpose of the restriction on the transitivity rule is an important one. It reduces the number of rewrite steps due to transitivity by taking advantage of scoping information. The second rule

2. if $A \in \text{LIVE}(M)$, and, for some $i$, $a_i \in \text{FV}(A)$ then

$$\begin{align*}
\text{let } \Gamma \{ f a = M \} & \rightarrow \text{let } \Gamma \{ f a = M \} \\
\text{in } C[f b] & \rightarrow \text{in } C[f b \land A[a := b]]
\end{align*}$$

allows us to unwind an atomic constraint. Note that at least one of the variables in $A$ must be bound by the abstraction. The restriction is there to prevent rewrite steps which would not be useful anyway. The two last rules are similar to the second rule but deal with unwinding in mutually recursive constraint abstractions. A key property of the rewrite rules is that they lead to equivalent constraint terms.

**Lemma 2.** If $M \mapsto N$ then $M \Leftrightarrow N$

The property is easy to argue for the transitivity rule. For the second rule it follows from the unwinding property (Lemma 1). The two last rules rely on similar unwinding properties for unwinding in mutually recursive constraint abstractions.

### 3.1 Normal Forms

Intuitively a constraint term is in normal form when the rules in figure have been applied exhaustively. But nothing stops us from performing rewrite steps which just add new copies of atomic constraints which are already in the constraint term. We can of course do this an arbitrary number of times creating a sequence of terms which are different but “essentially the same”. To capture this notion of essentially the same we define a congruence which equates terms which are equal up to copies of atomic constraints.

**Definition 5.** We define $\sim$ as the reflexive, transitive, symmetric and compatible closure of the following clauses.

$$(i)\ A \land A \sim A \quad (ii)\ M \land \top \sim M \quad (iii)\ \top \land M \sim M$$

$$(iv)\ \text{if } \text{FV}(A) \cap \text{CV}(C) = \emptyset \text{ and the hole in } C \text{ is live then } C[A] \sim C[\top] \land A$$

Rewriting commutes with $\sim$ so we can naturally extend $\rightarrow$ to equivalence classes of $\sim$. With the help of $\sim$ we can define the notion of a productive rewrite step $M \sim N$ which is a rewrite step which adds a new atomic constraint.
Definition 6. $M \rightsquigarrow N$ iff $M \rightarrow N$ and $M \not\rightleftarrows N$.

Finally we arrive at our definition of normal form up to productive rewrite steps.

Definition 7. $M$ is in normal form iff $M \not\rightsquigarrow$.

The main technical theorem in this paper is that when a constraint term with no free constraint abstraction variables is in normal form then the models of the constraint term are exactly characterised by the free live atomic constraints of the constraint term.

Theorem 1. If $M$ is in normal form and $\text{FAV}(M) = \emptyset$ then $\theta;\emptyset \models M$ iff $\theta \models \text{LIVE}(M)$

Given a constraint term $M$ and a fixed assignment of some of the variables we can find its least model as follows. First we find an equivalent constraint term $N$ in normal form. Then we extract the free live atomic constraints of the normal form which exactly characterises the models of $N$ and $M$. Since $\text{LIVE}(N)$ is just a set of atomic constraints we can then proceed with any standard method, such as computing the transitive closure. The proof of Theorem 1 can be found in appendix B. The key component of the proof is the application of two key properties of unwindings of normal forms. The first property is that normal forms are preserved by unwindings.

Lemma 3. If let $\Gamma \{ f \ a = M \} \ in \ C[f \ b]$ is in normal form then the unwinding let $\Gamma \{ f \ a = M \} \ in \ C[M[a := b]]$ is in normal form.

The lemma guarantees normal forms of arbitrary unwindings of a normal form which we need because of the coinductive definition of $\theta;\Gamma \models M$. The second property is that unwinding of a normal form does not change the free live atomic constraints of the constraint term.

Lemma 4. If let $\Gamma \{ f \ a = M \} \ in \ C[f \ b]$ is in normal form then

$$\text{LIVE(let } \Gamma \{ f \ a = M \} \ in \ C[f \ b]) = \text{LIVE(let } \Gamma \{ f \ a = M \} \ in \ C[M[a := b]])$$

3.2 Computing Normal Forms

Given a constraint term $M$, we need to compute an equivalent term in normal form. Our algorithm relies on a representation of equivalence classes of terms with respect to $\rightsquigarrow$ and computes sequences of the form

$$M_0 \rightsquigarrow M_1 \rightsquigarrow M_2 \rightsquigarrow \ldots.$$

The termination of the algorithm is ensured by the following result.

Lemma 5. There is no infinite sequence of the form given above.
Proof (Sketch). Let $n$ be the number of variables (free and bound) in $M_0$. Note that the number of variables remain constant in each step. Thus the number of unique atomic constraints that can be added to $M$ is bounded by $n^2$. Since every productive rewrite step introduces a new atomic constraint the number of steps is bounded by $n^2$.

When given a constraint term as input, our algorithm first marks all atomic constraints. These marked constraints can be thought of as a work list of atomic constraints to consider. The algorithm then unmarks the constraints one by one and performs all productive rewrite steps which only involve atomic constraints which are not marked. The new atomic constraints which are produced by a rewrite step are initially marked. The algorithm maintains the following invariant: the term obtained by replacing the marked terms with $\top$ is in normal form. The algorithm terminates with a normal form when no atomic constraints remain marked. The pseudo code for this algorithm is given below.

Algorithm 1
1. Mark all atomic constraints.
2. If there are no remaining marked constraints then stop otherwise pick a marked atomic constraint and unmark it.
3. Find all productive redexes which involve the unmarked constraint and perform the corresponding rewrite steps. Let the added atomic constraints be marked.
4. Go to step 2

3.3 Data Structures

The efficiency of the algorithm relies on maintaining certain data structures. In step 3 of the algorithm we use data structures such that we can solve the following two problems:

1. find all redexes we need to consider in time proportional to the number of such, and
2. decide in constant time whether a redex is productive.

We can solve the first problem if we maintain, for every existentially bound variable $b$,

- a list of all $a$ in scope at the point where $b$ is bound, such that $a \leq b$ is an unmarked atomic constraint in the term.
- a list of all $c$ in scope at the point where $b$ is bound, such that $b \leq c$ is an unmarked atomic constraint in the term.

With this information we can easily list all transitivity-redexes we need to consider in step 3 in time proportional to the number of redexes. When we unmark a constraint we can update the data structure in constant time.

For the second problem, to decide in constant time whether a redex is productive, we need to decide, in constant time, whether the atomic constraint to be added already exists in the term. We can achieve this by a $n$ times $n$ bit-matrix.
where \( n \) is the number of variables (free and bound) in the constraint term. If \( a \leq b \) is in the term then the entry in the matrix for \((a, b)\) is 1 and 0 otherwise. This is sufficient for the complexity argument in the next section but in practice we use a refined data structure which we describe in section 3.5.

### 3.4 Complexity

The cost of the algorithm is dominated by the operations performed by step 3 which searches for productive redexes. The cost is proportional to the number of redexes (productive or non-productive) considered and each redex in the final normal form is considered exactly once in step 3. Thus the cost of step 3 is proportional to the number of redexes in the final normal form. An analysis of the maximum number of redexes gives the following.

- The maximum number of transitivity-redexes is, for each existentially quantified variable \( a \), the square of the number of variables in scope at the point where \( a \) is bound.
- The maximum number of unwind-redexes is, for each variable \( a \) bound in a constraint abstraction \( f \), two times the number of variables in scope at the point where \( a \) is bound times the number of calls to \( f \).

A consequence of this analysis is the complexity result we are about to state. Let the skeleton of a constraint term be the term where all occurrences of atomic constraints, and the trivially true constraint have been removed. What remains are the binding occurrences of variables and all calls to constraint abstractions. Now, for a constraint term \( M \), let \( n \) be the size of the skeleton of \( M \) plus the number of free variables of \( M \). The complexity of the algorithm can be expressed in terms of \( n \) as follows.

**Theorem 2.** The normal form can be computed \( O(n^3) \) time.

### 3.5 Refined Data Structure

The cost of initialising the bit-matrix described in section 3.3 is dominated by the cost of step 3 in the algorithm but we believe that in practice the cost of initialising the matrix may be significant. Also the amount of memory required for the matrix is quite substantial and many entries in the matrix would be redundant since the corresponding variables have no overlapping scope. Below we sketch a refined approach based on this observation which we believe will be important in practice. We associate a natural number, index("a"), with every variable "a". We assign the natural number as follows. First we choose an arbitrary order for all the free variables and bind them existentially, in this order, at top level. Then we assign to each variable the lexical binding level of the variable. For example, in \( \exists a. (\exists b. M) \land (\exists c. N) \) we assign 0 to \( a \), 1 to \( b \) and \( c \), and so on. Note that the number we assign to each variable is unique within the scope of the variable. Given this we have the following data structures. For every variable \( b \),
– a set of all \( a \) such that \( \text{index}(a) \leq \text{index}(b) \) and \( a \leq b \) is an atomic constraint (marked or unmarked) in the term.
– a set of all \( c \) such that \( \text{index}(c) \leq \text{index}(b) \) and \( b \leq c \) is an atomic constraint (marked or unmarked) in the term.

The sets have, due to scoping, the property that, for any two distinct elements \( a \) and \( b \), \( \text{index}(a) \) is distinct from \( \text{index}(b) \). Thus the sets can be represented by bit-arrays, indexed by \( \text{index}(a) \) so that set membership can be decided in constant time. Now, to decide whether an atomic constraint \( a \leq b \) is in the constraint becomes just set membership in the appropriate set.

4 Related Work

The motivation for this paper is to reduce the cost of the combination of subtyping and polymorphism and in this respect it is related to numerous papers on constraint simplification techniques \[ \text{FM89, Cmr90, Kae92, Smi94, EST95, Pot96, TS96, FA96, AWP97, Reh97, EF97}. \] Our work is particularly related to the work by Dussart, Henglein and Mossin on binding-time analysis with binding-time-polymorphic recursion \[ \text{DHM95} \] where they use constraint simplification techniques in combination with a clever fixed-point iteration to obtain a polynomial time algorithm. In his thesis Mossin applied these ideas to show that a flow analysis with flow-polymorphic recursion can be implemented in polynomial time \[ \text{Mos97} \]. Our flow analysis in appendix A that we give as an example of how constraint abstractions can be used, is based on this flow analysis. A consequence of the complexity of our constraint solving algorithm is that the analysis can be implemented in \( O(n^3) \) time where \( n \) is the size of the explicitly type program. This is a substantial improvement over the algorithm by Mossin which is \( O(n^8) \) \[ \text{EF97} \].

To represent instantiation in the constraint language is not a new idea. It goes back at least to Henglein’s work on type-polymorphic recursion \[ \text{Hen93} \] where he uses semiumification constraints to represent instantiation. Although constraint abstractions and semiumification constraints may have similar applications they are inherently different: Semiumification constraints are inequality constraints of the form \( A \leq B \) which constrains the (type) term \( B \) to be an instance of \( A \) by an unknown substitution. In contrast, a call to a constraint abstraction denotes a given instance of the constraints in the body of the abstraction.

Closely related to our work is the recent work by Rehof and Fähndrich \[ \text{RF01} \] where they also give an \( O(n^3) \) algorithm for Mossin’s flow analysis. The key idea in their and our work is the same – to represent substitution instantiation in the constraints by extending the constraint language. However, the means are

\[ ^2 \text{In his thesis Mossin states that he believes that the given algorithm can be improved. In fact an early version of [DHM95] contained a } O(n^3) \text{ algorithm for binding-time analysis but it was removed from the final version since its correctness turned out to be non-trivial (personal communication with Fritz Henglein).} \]
not the same. Where we use constraint abstractions they use \textit{instantiation constraints}, a form of inequality constraints similar to semiunification constraints but labelled with an instantiation site and a polarity. They compute the flow information from the constraints through an algorithm for \textit{Context-Free Language (CFL) reachability} \cite{Rep97, MR00}. A key difference between constraint abstractions and instantiation constraints is that constraint abstractions offer more structure and a notion of local scope whilst in the work by Rehof and Fähndrich all variables scope over the entire set of constraints. Our algorithm takes advantage of the scoping in an essential way. Firstly, we do not add any edges between variables that have no common scope and secondly the scoping comes into the restriction of our transitivity rule and the unwind rules. Although the scoping does not improve the asymptotic complexity in terms of the size of the explicitly typed program it shows up in the more fine-grained complexity argument leading to the cubic bound (see section 3.4) and it is essential for the refined data structures we sketch in section 3.5. Constraint abstractions also offer a more subjective advantage – the additional structure of constraint abstractions enforces many useful properties. As a result we think it will be easy to use constraint abstractions in a wide range of type based analyses and we think that constraint abstractions will not lead to any additional difficulties when establishing the soundness of the analyses.

We have previously used constraint abstraction to formulate usage-polymorphic usage analyses with usage subtyping \cite{Sve00, GS00} and an algorithm closely related to the one in this paper is presented in the second author’s masters thesis \cite{Sve00} (\cite{GS00} focuses on the usage type system and no constraint solving is presented).

5 Conclusions and Future Work

In this paper we have shown how a constraint language with simple inequality constraints over a lattice can be extended with constraint abstractions which allow the constraints to compactly express substitution instantiation. The main result of this paper is a constraint solving algorithm which computes least solutions to the extended form of constraints in cubic time. In \cite{GS00} we have used this expressive constraint language to formulate a usage-polymorphic usage analyses with usage subtyping and usage-polymorphic recursion and in an appendix to this paper we demonstrate how the extended constraint language can be used to yield a cubic algorithm for Mossin’s polymorphic flow analysis with flow subtyping and flow polymorphic recursion \cite{Mos97}. We believe that our approach can be applied to a number of other type based program analyses such as binding time analyses, e.g., \cite{DHM95}, points-to-analyses, e.g., \cite{FFA00} and uniqueness type systems \cite{BS96}.

An interesting possibility for future work is to explore alternative constraint solving algorithms. The current algorithm has a rather compositional character in that, it rewrites the body of a constraint abstraction without considering how it is called. In \cite{Sve00} we describe an algorithm where the different calls to a
constraint abstraction lead to rewrites inside the abstraction. The algorithm can in this way take advantage of global information (it can be thought of as a form of caching) which yields a interesting finer grained complexity characterisation. The algorithm in [Sve00] is however restricted to non-recursive constraint abstractions and it is not clear whether the algorithm can be extended to recursive constraint abstractions (although we believe so). Another opportunity for future work is to investigate whether constraint abstractions can be a useful extension for other underlying constraint languages. Constraint abstraction could also possibly be made more powerful by allowing constraint abstractions to be passed as parameters to constraint abstractions (i.e., making them higher order). Finally a practical comparison with Mossin’s algorithm and the algorithm by Rehof and Fähndrich remains to be done. The outcome of such a comparison is not clear to us.

Acknowledgements. We would like to thank our supervisor David Sands for his support and the anonymous referees for their useful comments.

References


A Flow Analysis

In this appendix we illustrate how constraint abstractions can be used in practice. As an example, we briefly present a flow-polymorphic type based flow analysis with flow-polymorphic recursion. For another example see [GS00] where constraint abstractions are used in usage analysis. The flow analysis is based on the flow analysis by Mossin [Mos97] but we use our extended constraint language with constraint abstractions. A similar analysis, but without polymorphic recursion, is given by Faxén [Fax95]. For simplicity we restrict ourself to a simply
typed functional language. To extend the analysis to a language with a Hindley-Milner style type system is not difficult. See for example [Fax95]. A key result is that the analysis can be implemented in $O(n^3)$ time where $n$ is the size of the explicitly typed program which is a substantial improvement over the algorithm by Mossin which is $O(n^8)$ [Mos97].

The aim of flow analysis is to statically compute an approximation to the flow of values during the execution of a program. To be able to pose flow questions we will label subexpressions with unique flow variables. We will label expressions in two distinct ways, as a source of flow or as a target of flow. We will use $e^a$ as our notation for labelling $e$ (with flow variable $a$) as a source of flow and $e_a$ as our notation for labelling $e$ as a target of flow. If we are interested in the flow of values from producers to consumers then we label all program points where values are created as sources of flow, we label the points where values are destructed as targets of flow, and we leave all other subexpressions unlabelled. In the example below we have labelled all values as sources with flow variables $a_0$ through $a_4$ and we have labelled the arguments to plus as targets with $a_5$ and $a_6$. We have not labelled the others consumers (the applications) to keep the example less cluttered.

\[
\begin{align*}
\text{let } \text{apply} &= (\lambda f. (\lambda y. f \, y)^{a_0})^{a_1} \\
\text{in let } \text{id} &= (\lambda x. x)^{a_2} \\
\text{in } (\text{apply id} \, 5^{a_3})_{a_5} + (\text{apply id} \, 7^{a_4})_{a_6}
\end{align*}
\]

We may now ask the question “which values may show up as arguments to plus?”. Our flow analysis will give the answer that the value labelled with $a_3$ (5) may flow to $a_5$ (the first argument) and $a_4$ (7) may flow to $a_6$ (the second argument). In this example the flow polymorphic types that we assign to $id$ and $apply$ plays a crucial role. A monomorphic system would conservatively say that both values could flow to both places. For some applications we might be interested in, not only the flow from producers to consumers, but also the flow to points on the way from a consumer to a producer. In our example we might be interested in the flow to $x$ in the body of $id$. We then add a target label on $x$ as in

\[
\begin{align*}
\text{let } \text{apply} &= (\lambda f. (\lambda y. f \, y)^{a_0})^{a_1} \\
\text{in let } \text{id} &= (\lambda x. x)^{a_2} \\
\text{in } (\text{apply id} \, 5^{a_3})_{a_5} + (\text{apply id} \, 7^{a_4})_{a_6}
\end{align*}
\]

and then ask for the flow to $a_7$. Our analysis would answer with $a_3$ and $a_4$. An important property of the analysis is that the type of $id$ remains polymorphic even though we tap off the flow passing through $x$. Thus our type system corresponds to the sticky interpretation of a type derivation in [Mos97]. The key to this property is to distinguish between source labels and target labels. If the label on $x$ would serve as both a source and a target label the flow through $id$ would be monomorphic.\footnote{We can achieve this degrading effect by annotating $x$ both as a source and as a target but using the same flow variable, i.e., as $x_{a_7}^{a_7}$.}
The language we consider is a lambda calculus extended with recursive let-expressions, integers, lists and case-expressions. The grammar of the language is as follows.

Variables \( x, y, z \)

Flow Variables \( a \)

Expressions \[ e ::= \lambda x.e \mid n \mid \text{nil} \mid \text{cons} e_0 e_1 \mid x \mid e_0 + e_1 \mid e_0 e_1 \mid \text{let} \{ b \} \text{in} e \mid \text{case} e \text{ of} alts \mid e^a \mid e_a \]

Bindings \( b ::= x = e \)

Alternatives \( alts ::= \{ \text{nil} \Rightarrow e_0, \text{cons} x y \Rightarrow e_1 \} \)

The language is simply typed and for our complexity result we assume that the terms are explicitly typed by having type annotations attached to every subterm. For our flow analysis we label the types of the underlying type system with flow variables.

Flow Types \( \tau ::= \text{Int}^a \mid (\tau \rightarrow \tau')^a \mid (\text{List} \tau)^a \)

We will let \( \rho \) range over flow types without the outermost annotation. The subtype entailment relation which take the form \( M \vdash \tau_0 \leq \tau_1 \) is defined in Figure 2. Recall that \( M \) ranges over constraint terms as defined in Section 2.1.

We read \( M \vdash \tau_0 \leq \tau_1 \) as “from the constraint term \( M \) it can be derived that \( \tau_0 \leq \tau_1 \)”. We will let \( \sigma \) range over type schemas.

Type Schemas \( \sigma ::= \forall a. f a \Rightarrow \tau \)

Since the underlying type system is monomorphic type schemas will only quantify over flow variables. A type schema contains a call \( f a \) to a constraint abstraction which may constrain the quantified variables. We will let \( \Theta \) and \( \Delta \) range over typing contexts which associates variables with types or type schemas depending on whether it is a let-bound variable or not. We will use juxtaposition as our notation for combining typing contexts. Our typing judgements take the form \( \Theta ; M \vdash e : \tau \) for terms, \( \Theta ; F \vdash b : (x : \sigma) \) for bindings and \( \Theta ; \Gamma \vdash \{ b \} : \Delta \) for groups of bindings. (Recall that \( F \) ranges over constraint abstractions and that \( \Gamma \) ranges over sets of constraint abstractions.) The typing rules of the analysis can be seen in Figure 3. The key difference to the type system in [M97] is in the rule Binding where generalisation takes place. Instead of putting the constraint term used to type the body of the binding into the type schema the constraint term is inserted into a new constraint abstraction and a call to this abstraction is included in the type schema.

To compute the flow in a program we can proceed as follows. First we compute a principal typing of the program which includes a constraint term where the free variables are the flow variables labelling the program. We then apply the algorithm from Section 3 and extract a set of atomic constraints which we can view as a graph. If there is a path from \( a_0 \) to \( a_1 \) then \( a_0 \) may flow to \( a_1 \). The typing rules as presented here are not syntax directed and cannot directly be interpreted as describing an algorithm for computing principal typings. Firstly,
\[ \begin{align*}
\top \vdash \text{Int} \leq \text{Int} & \quad M \vdash \tau \leq \tau' \\
M \vdash \tau_0' \leq \tau_0 & \quad N \vdash \tau_1' \leq \tau_1' \\
M \land N \vdash \tau_0 \rightarrow \tau_1 \leq \tau_0' \rightarrow \tau_1' & \quad M \vdash \rho_0 \leq \rho_1 \\
M \land (a \leq a') \vdash (\text{List} \tau)^a \leq (\text{List} \tau')^{a'} &
\end{align*} \]

Fig. 2. Subtyping rules

\[ \begin{align*}
\text{Abs} & \quad \begin{array}{c} \Theta \vdash \{x : \tau\} ; M \vdash e : \tau' \\
\Theta ; M \vdash \lambda x . e : (\tau \rightarrow \tau')^a & \end{array} & \text{Int} & \quad \begin{array}{c} \top \vdash n : \text{Int}^a \\
\Theta ; \top \vdash \text{nil} : (\text{List} \tau)^a & \end{array} & \text{Nil} & \quad \begin{array}{c} \Theta ; \top \vdash \text{nil} : (\text{List} \tau)^a \\
\Theta ; M \vdash e_0 : \tau & \quad \Theta ; N \vdash e_1 : (\text{List} \tau)^a & \Theta ; M \land N \vdash \text{cons} e_0 e_1 : (\text{List} \tau)^a & \end{array} \\
\text{Cons} & \quad \begin{array}{c} \Theta ; M \vdash e_0 : \tau \\
\Theta ; N \vdash e_1 : (\text{List} \tau)^a & \Theta ; M \land N \vdash \text{cons} e_0 e_1 : (\text{List} \tau)^a & \end{array} \\
\text{Var-} & \quad \begin{array}{c} \Theta \vdash \{x : \forall a . f a \Rightarrow \tau\} ; f b \vdash x : \tau[a := b] \\
\varnothing \vdash x : \tau & \end{array} & \text{Var-} & \quad \begin{array}{c} \Theta \vdash \{x : \tau\} ; \top \vdash x : \tau \\
\varnothing \vdash x : \tau & \end{array} \\
\text{Plus} & \quad \begin{array}{c} \Theta ; M \vdash e_0 : \text{Int}^a_0 \\
\Theta ; N \vdash e_1 : \text{Int}^{a_1} & \Theta ; M \land N \vdash e_0 + e_1 : \text{Int}^a & \end{array} \\
\text{App} & \quad \begin{array}{c} \Theta ; M \vdash e_0 : (\tau \rightarrow \tau')^a \\
\Theta ; N \vdash e_1 : \tau & \Theta ; M \land N \vdash e_0 e_1 : \tau' & \end{array} \\
\text{Case} & \quad \begin{array}{c} \Theta ; M \vdash e : \tau \\
\Theta ; N \vdash \text{als} : \tau \Rightarrow \tau' & \Theta ; M \land N \vdash \text{case e of als : } \tau' & \end{array} \\
\text{Let} & \quad \begin{array}{c} \Theta \Delta ; \Gamma \vdash \{b\} : \Delta \\
\Theta ; \Delta ; M \vdash e : \tau \\
\Theta ; \text{let } \Gamma \text{ in } M \vdash \text{let } \{b\} \text{ in } e : \tau & \end{array} & \text{Alts} & \quad \begin{array}{c} \Theta \Delta ; \Gamma \vdash \{b\} : \Delta \\
\Theta ; \Delta ; M \vdash \{\text{nil} \Rightarrow e_0 ; \text{cons } x y \Rightarrow e_1\} : (\text{List} \tau)^a & \Theta ; M \land N \vdash \{\text{nil} \Rightarrow e_0 ; \text{cons } x y \Rightarrow e_1\} : (\text{List} \tau)^a \Rightarrow \tau' & \end{array} \\
\text{Source} & \quad \begin{array}{c} \Theta ; M \vdash e : \rho^a \\
\Theta ; M \land (a \leq c) \land (b \leq c) \vdash e^b : \rho^c & \end{array} & \text{Target} & \quad \begin{array}{c} \Theta ; M \vdash e : \rho^a \\
\Theta ; M \land (a \leq c) \land (a \leq b) \vdash e_b : \rho^c & \end{array} \\
\text{Binding group-} & \quad \begin{array}{c} \Theta ; \emptyset \vdash \emptyset : \emptyset \\
\Theta ; M \vdash e : \tau & \end{array} & \text{Binding group} & \quad \begin{array}{c} \Theta ; \Gamma \vdash \{b\} : \Delta \\
\Theta ; F \vdash b : (x : \sigma) & \Theta ; \Gamma[F] \vdash \{b, b\} : (\Delta, x : \sigma) & \end{array} \\
\text{Binding} & \quad \begin{array}{c} \Theta ; f a = M \vdash x = e : (x : \forall a . f a \Rightarrow \tau) \\
\{a\} \cap \text{FV}(\Theta, f a = M, e) = \emptyset & \end{array} \\
\text{Sub} & \quad \begin{array}{c} \Theta ; M \vdash e : \tau \\
N \vdash \tau \leq \tau' & \Theta ; \exists a . M \vdash e : \tau & \{a\} \cap \text{FV}(\Theta, e, \tau) = \emptyset & \end{array}
\end{align*} \]

Fig. 3. Typing rules for a flow analysis
the subsumption rule (Sub) and the rule (Exist-intro) which introduces existential quantification in constraints can be applied everywhere in a typing derivation. This problem is solved by the standard approach to incorporate (Sub) and (Exists-intro) into an appropriate subset of the other rules to obtain a syntax-directed set of rules. Secondly, in the rule (Let) an inference algorithm would have to come up with an appropriate $\Delta$. However, this only amounts to coming up with fresh names: Clearly, $\Delta$ would have to contain one type associations of the form $x : \sigma$ for each variable defined by the let-expression. Recall that $\sigma$ is of the form $\forall a . f a \Rightarrow \tau$. We obtain $\tau$ simply by annotating the underlying type with fresh flow variables. Since they are fresh we will be able to generalise over all of them so we can take $a$ to be these variables in some order. Finally we generate the fresh name $f$ for the constraint abstraction. Note that no fixed-point calculation is required which is possible because we have recursive constraint abstractions. Now let us apply the algorithm to our example program. We first compute the constraint term in the principal typing which yields the following.

\[
\begin{align*}
\text{let } & f \text{apply } b_0 b_1 b_2 b_3 b_4 b_5 b_6 = \exists c_0 . \exists c_1 . \exists c_2 . (b_3 \leq b_0) \land (b_1 \leq b_4) \land (b_2 \leq c_2) \land (c_1 \leq b_5) \land (a_0 \leq b_5) \land (c_0 \leq b_6) \land (a_1 \leq b_6) \\
\text{in let } & f \text{id } b_0 b_1 b_2 = \exists c_0 . \exists c_1 . (b_0 \leq c_1) \land (c_1 \leq b_1) \land (c_1 \leq a_7) \land (c_0 \leq b_2) \land (a_2 \leq b_2) \\
\text{in } & \exists c_0 . . . \exists c_{18} . (f \text{apply } c_0 c_1 c_2 c_3 c_4 c_5 c_6) \land (f \text{id } c_0 c_1 c_2) \land (c_7 \leq c_3) \land (a_3 \leq c_3) \land (c_4 \leq c_8) \land (c_4 \leq a_5) \land (f \text{apply } c_{10} c_{11} c_{12} c_{13} c_{14} c_{15} c_{16}) \land (f \text{id } c_{10} c_{11} c_{12}) \land (c_{17} \leq c_{13}) \land (a_3 \leq c_{13}) \land (c_{14} \leq c_{18}) \land (c_{14} \leq a_5)
\end{align*}
\]

Then we apply the algorithm from Section 3 and extract the set of free live atomic constraints which is \{ $a_3 \leq a_5$, $a_4 \leq a_6$, $a_3 \leq a_7$, $a_4 \leq a_7$ \}. The paths in this constraint set (viewed as a graph) is the result of the analysis.

Finally, by inspecting the rules we can see that the size of the skeleton of the constraint term required to type a program is proportional to the size of the explicitly typed program and that the number of free variables is the number of flow variables in the program. From this fact and theorem 4 we can conclude that the complexity of the flow analysis is $O(n^3)$ where $n$ is the size of the typed program.

## B Proof of Theorem 1

In this appendix we give a proof of Theorem 1. We first introduce a form of constraint term contexts, reminiscent of evaluation contexts used in operational semantics, where the hole may not occur under any binder.

Evaluation Contexts $E ::= [\cdot] \mid E \land M \mid M \land E$

Note that the hole in an evaluation context is always live. We have the following properties for evaluation contexts which we state without proof.

**Lemma 6.** 1. Let $\Gamma$ in $E[\text{let } \Gamma' \text{ in } M]$ is in normal form iff let $\Gamma \Gamma'$ in $E[M]$ is in normal form.
2. \( \text{LIVE}(\text{let } \Gamma \text{ in } E[\text{let } \Gamma' \text{ in } M]) = \text{LIVE}(\text{let } \Gamma \Gamma' \text{ in } E[M]) \).

3. If \( a \notin \text{FV}(\Gamma, E) \), and \( \text{let } \Gamma \text{ in } E[\exists a. M] \) is in normal form then \( \text{let } \Gamma \text{ in } E[M] \) is in normal form.

The key to the proof of Theorem 1 is the following auxiliary relation.

**Definition 8.** We define an auxiliary relation \( \theta; \Gamma \models M \) as:

1. \( \text{let } \Gamma \text{ in } E[M] \) is in normal form,
2. \( \theta \models \text{LIVE}(\text{let } \Gamma \text{ in } E[M]) \),
3. \( \text{FAV}(\text{let } \Gamma \text{ in } E[M]) = \emptyset \).

The technical core of the proof now shows up in the proof of the following lemma.

**Lemma 7.** if \( \theta; \Gamma \models M \) then \( \theta; \emptyset \models M \).

Before we proceed with the proof of this lemma we will use it to establish Theorem 1.

*Proof (Theorem 1).* Assume the premise. The right way implication (if \( \theta; \emptyset \models M \) then \( \theta \models \text{LIVE}(M) \)) follows the fact that all syntactically live constraints are semantically live. To show the left way implication (if \( \theta \models \text{LIVE}(M) \) then \( \theta; \emptyset \models M \)) assume that \( \theta \models \text{LIVE}(M) \) which immediately gives \( \theta; \emptyset \models M \). Thus, by Lemma 7, \( \theta; \emptyset \models M \) as required.

Finally we prove Lemma 7.

*Proof (Lemma 7).* Recall that \( \theta; \Gamma \models M \) is defined coinductively by the rules in Figure 1. That is, \( \models \) is defined as the largest fixed point of the functional \( \mathcal{F} \) expressed by the rules. By the coinduction principle we can show that \( \models \subseteq \models \) if we can show that \( \models \subseteq \mathcal{F}(\models) \). Thus we assume that \( \theta; \Gamma \models M \) and proceed by case analysis on the structure of \( M \).

**case** \( M \equiv a \leq b \): By the definition of \( \theta; \Gamma \models a \leq b \) there exists \( E \) which fulfils the requirements in Definition 8. In particular, \( \theta \models \text{LIVE}(\text{let } \Gamma \text{ in } E[a \leq b]) \). Since \( E \) cannot capture variables and the hole in \( E \) is live we know that \( a \leq b \in \text{LIVE}(\text{let } \Gamma \text{ in } E[M]) \) so \( \theta; \Gamma \mathcal{F}(\models) a \leq b \).

**case** \( M \equiv \top \): Trivial.

**case** \( M \equiv K \land L \): To show that \( \theta; \Gamma \mathcal{F}(\models) K \land L \) we need to show that \( \theta; \Gamma \models K \) and \( \theta; \Gamma \models L \). We will only show the former, the latter follows symmetrically. By the definition of \( \theta; \Gamma \models K \land L \) there exists \( E \) which fulfils the requirements in Definition 8. Take \( E' \) to be \( E[\cdot] \land L \). Then \( E' \) is a witness of \( \theta; \Gamma \models K \).
case \( M \equiv \text{let } \Gamma' \text{ in } N \): We may without loss of generality (due to properties of \( \alpha \)-conversion) assume that the constraint abstraction variables defined \( \Gamma \) and \( \Gamma' \) are disjoint. To show that \( \theta; \Gamma \models F(\cdot) =) \text{ let } \Gamma' \text{ in } N \) we need to show that \( \theta; \Gamma \models N \). By the definition of \( \theta; \Gamma \models \text{let} \Gamma' \text{ in } M \) there exists \( E \) which fulfils the requirements in Definition 8. Floating of let bindings preserves normal forms (Lemma 8) so we can float out \( \Gamma' \) and obtain \( \text{let } \Gamma \Gamma' \text{ in } E[M] \) in normal form. Also, by Lemma 9 \( \text{LIVE(\text{let } \Gamma \text{ in } E[\text{let } \Gamma' \text{ in } M]) = \text{LIVE(\text{let } \Gamma \Gamma' \text{ in } E[M])} \). Thus \( E \) is a witness of \( \theta; \Gamma \Gamma' \models M \).

case \( M \equiv f b \): By the definition of \( \theta; \Gamma \models f b \) we know that \( f \) must bound by \( \Gamma \), i.e., \( \Gamma = \Gamma'\{f a = N\} \) for some \( \Gamma' \) and some \( N \). We are required to show that \( \theta; \Gamma'\{f a = N\} \models N[a := b] \). By the definition of \( \theta; \Gamma \models f b \) we know that there exists \( E \) which fulfils the requirements in Definition 8. Without loss of generality (due to properties of \( \alpha \)-conversion) assume that \( a \notin \text{FV}(\Gamma, E) \). Since \( \text{let } \Gamma \text{ in } E[\exists a.N] \) is in normal form, \( \text{let } \Gamma \text{ in } E[N[a := b]] \) is normal form. Also, by Lemma 8 \( \text{LIVE(\text{let } \Gamma \text{ in } E[f a = N]) = \text{LIVE(\text{let } \Gamma' \text{ in } E[N[a := b]])} \). Thus \( E \) is a witness of \( \theta; \Gamma'\{f a = N\} \models N[a := b] \).

case \( M \equiv \exists a.N \) To show that \( \theta; \Gamma \text{ F(\cdot) } \exists a.N \) we need to show that there exists \( d \in \mathcal{I} \) such that \( \theta[a := d]; \Gamma \models N \). Let

\[
\begin{align*}
    d & = \bigcup \{ \theta(a') \mid a' \neq a \text{ and } a' \leq a \in \text{LIVE}(N) \}.
\end{align*}
\]

By the definition of \( \theta; \Gamma \models \exists a.N \) there exists \( E \) which fulfils the requirements in Definition 8. Without loss of generality (due to properties of \( \alpha \)-conversion) we can assume that \( a \notin \text{FV}(\Gamma, E) \). Since \( \text{let } \Gamma \text{ in } E[\exists a.N] \) is in normal form, \( \text{let } \Gamma \text{ in } E[N[a := b]] \) is in normal form. It remains to show that \( \theta[a := d] \models \text{LIVE(\text{let } \Gamma \text{ in } E[N])} \). Given \( A \in \text{LIVE(\text{let } \Gamma \text{ in } E[N])} \) we proceed by the following cases.

subcase \( A \equiv a \leq a \): Trivial.

subcase \( A \equiv b \leq c \text{ where } b \neq a \text{ and } c \neq a \): In this case \( A \in \text{LIVE(\text{let } \Gamma \text{ in } E[\exists a.N])} \) so \( \theta \models A \) and thus \( \theta[a := d] \models A \).

subcase \( A \equiv b \leq a \text{ where } b \neq a \): In this case \( A \in \text{LIVE}(N) \) and thus \( \theta[a := d] \models A \) by the construction of \( d \).

subcase \( A \equiv a \leq b \text{ and } b \neq a \): In this case \( a \leq b \in \text{LIVE}(N) \). We will show that \( \theta(b) \) is an upper bound of

\[
\{ \theta(a') \mid a' \neq a \text{ and } a' \leq a \in \text{LIVE}(N) \}
\]

and, since \( d \) is defined as the least upper upper bound, \( \theta[a := d] \models a \leq b \) follows. Now given any \( a' \) such that \( a' \neq a \) and \( a' \leq a \in \text{LIVE}(N) \). Since
\[
\begin{align*}
a' \leq a \in \text{LIVE}(N) \text{ and } a \leq b \in \text{LIVE}(N) \text{ we know that } \text{let } \Gamma \text{ in } E[\exists a. N] \rightarrow \text{let } \Gamma \text{ in } E[\exists a. N \land a' \leq b] \text{ and since let } \Gamma \text{ in } E[\exists a. N] \text{ is in normal form we know that} \\
\text{LIVE}(\text{let } \Gamma \text{ in } E[\exists a. N \land a' \leq b]) = \text{LIVE}(\text{let } \Gamma \text{ in } E[\exists a. N]).
\end{align*}
\]

Finally, since \(a' \neq a\) it must be the case that \(a' \leq b \in \text{LIVE}(\text{let } \Gamma \text{ in } E[\exists a. N \land a' \leq b])\) and thus \(a' \leq b \in \text{LIVE}(\text{let } \Gamma \text{ in } E[\exists a. N])\). Hence \(\theta \models a' \leq b\) so \(\theta(a') \subseteq \theta(b)\).
Implementing Circularity Using Partial Evaluation

Julia L. Lawall

DIKU, University of Copenhagen
Universitetsparken 1
DK-2100 Copenhagen, Denmark
julia@diku.dk

Abstract. Complex data dependencies can often be expressed concisely by defining a variable in terms of part of its own value. Such a circular reference can be naturally expressed in a lazy functional language or in an attribute grammar. In this paper, we consider circular references in the context of an imperative C-like language, by extending the language with a new construct, persistent variables. We show that an extension of partial evaluation can eliminate persistent variables, producing a staged C program. This approach has been implemented in the Tempo specializer for C programs, and has proven useful in the implementation of run-time specialization.

1 Introduction

In compilation and program transformation, the treatment of a subcomponent of a block of code often depends on some global properties of the code itself. A compiler needs to know whether the source program ever uses the address of a local variable, to decide whether the variable must be allocated on the stack [1]. A partial evaluator needs to know whether a dynamic (but non-side-effecting) expression is referred to multiple times, to decide whether the expression should be named using a let expression [4, 23]. In the context of run-time specialization, we have found that optimizing the specialized code based on its total size and register usage can significantly improve its performance [15]. Such programs can often be efficiently implemented as multiple phases, where early phases collect information and later phases perform the transformation. This organization, however, distributes the treatment of each subcomponent of the input across the phases, which can introduce redundancy, and complicate program understanding and maintenance.

In a lazy language, we can implement these examples in a single traversal using recursive declarations, as investigated by Bird [3]. The canonical example of such a circular program is “repmin,” which reconstructs a tree, such that the value at each leaf is the least value at any leaf in the original tree. While repmin

* This research was partially supported by the Danish Natural Science Research Council (PLT project).
can be implemented by first traversing the tree to detect the least value, and
then traversing the tree again to construct the result, the program can also be
expressed in a lazy language as follows [3]:

```haskell
data Tree = Tip Int | Fork Tree Tree

rm t = fst p
  where p = repmin t (snd p)

repmin (Tip n) m = (Tip m, n)

repmin (Fork L R) m = (Fork t1 t2, min m1 m2)
  where (t1, m1) = repmin L m
       and (t2, m2) = repmin R m
```

The variable `p` in `rm` represents a value that is the result of the traversal of the
tree, but that is used in computing that result as well. Here, a lazy evaluation
strategy suffices to order the computations such that the components of `p` are
determined before they are used.

Nevertheless, the use of a lazy language is not always appropriate. To resolve
this dilemma, we propose a language extension, *persistent variables*, that can
describe circular references in an imperative language. Using this facility, we can
implement `repmin` imperatively as follows, where the persistent variable `minval`
implements the circular reference to `p` in the functional implementation:

```c
typedef struct ans {
  int mn;
  Tree *tree;
} Ans;

Tree *rm(Tree *t) {
  persistent int minval;
  Ans a;
  repmin(t, pread(minval), &a);
  pwrite(minval,a.mn);
  return a.tree;
}

void repmin(Tree *t, int m, Ans *a) {
  Ans a1, a2;
  if (t->type == Fork) {
    repmin(t->left, m, &a1);
    repmin(t->right, m, &a2);
    a->mn = min(a1.mn,a2.mn);
    a->tree = mkFork(a1.tree,a2.tree);
  } else /* (t->type == Tip) */ {
    a->mn = t->tipval;
    a->tree = mkTip(m);
  }
}
```

This implementation uses `pread` ("persistent read") to reference the final value
to which `minval` is initialized using `pwrite` ("persistent write"). To execute the
program, we must first transform it such that it initializes `minval` before any
reference. In this paper, we show how to use partial-evaluation technology to
perform this transformation.

Traditionally, partial evaluation specializes a program with respect to a sub-
set of its inputs. The program is evaluated in two stages: The first stage per-
forms the *static* computations, which depend only on the known input. When

---
1 The structure *Ans* is used to return multiple values from the function *repmin*. 
the rest of the input is available, the second stage evaluates the remaining dynamic computations, producing the same result as the original program. With minor extensions, we can use this framework to eliminate circularity by simply considering computations that depend on the persistent variables to be dynamic, and the other computations to be static. For example, in the above implementation of repmin, the construction of each leaf of the output tree depends on the value of the persistent variable minval, via the parameter m of repmin, and is thus dynamic. The calculation of the least value in the tree depends only on the input tree, and is thus static. The staging performed by partial evaluation permits the persistent variables that only depend on static information to be initialized in the static phase and read in the dynamic phase. We have implemented this approach in the Tempo partial evaluator for C programs, developed in the Compose group at IRISA [6].

This implementation of circularity leads naturally to incremental specialization [7,17]; if the value of a persistent variable depends on that of another persistent variable, partial evaluation must be iterated. If there are recursive dependencies among the persistent variables, however, the program cannot be treated by our approach (cf. Section 6.3).

The rest of this paper is organized as follows. Section 2 describes partial evaluation in more detail. Section 3 presents the implementation of persistent variables in the context of a partial evaluator for a simple imperative language. Section 4 gives a semantics of the language with persistent variables and shows that partial evaluation of a program preserves its semantics. Section 5 compares our partial evaluation-based approach to related techniques in the implementation of attribute grammars. Section 6 provides some examples of the use of persistent variables. Section 7 describes related work, and Section 8 concludes.

2 Specialization Using Partial Evaluation

Partial evaluation uses interprocedural constant propagation to specialize a program with respect to some of its inputs. We use offline partial evaluation, in which each expression is annotated as static or dynamic by a preliminary binding-time analysis phase. Two kinds of specialization can be performed in this framework: program specialization and data specialization. Program specialization transforms a program into an optimized implementation based on the results of evaluating the static subexpressions [11]. Static subexpressions are replaced by their values, static conditionals are reduced, and static loops are unrolled. Data specialization separates a program into two stages, known as the loader and the reader, before the static data is available [2]. The loader stores the values of the static subexpressions in a data structure known as a cache. The reader has the form of the source program, but with the static subexpressions replaced by cache references. Because the loader and reader are independent of the static data, conditionals are not reduced and loops are not unrolled.

We implement persistent variables in the context of data specialization. Persistent variables and data specialization fit together well, because the data spe-
cialization cache is a natural means to transmit the values of persistent variables from the static phase to the dynamic phase.

3 Data Specialization and Persistent Variables

We now define data specialization for a simple imperative language with persistent variables. Treating a richer language is straightforward; the implementation allows the full subset of C accepted by Tempo [6], including pointers and recursive functions.

3.1 Source Language

A program consists of declarations $d$ and a statement $s$, defined as follows:

$$
\begin{align*}
\mathit{d} & \in \mathit{Declaration} ::= \text{int } x | \text{persistent int } p \\
\mathit{s} & \in \mathit{Statement} ::= x = e | \text{pwrite}(p, e) | \text{if}(e) \mathit{s}_1 \text{ else } \mathit{s}_2 \\\n& \hspace{1cm} | \text{while}(e) \mathit{s} \{\mathit{s}_1; \ldots; \mathit{s}_n\} \\
\mathit{e} & \in \mathit{Expression} ::= c | x | \mathit{e}_1 \text{ op } \mathit{e}_2 | \text{pread}(p) \\
\mathit{x} & \in \mathit{Variable} \\
\mathit{p} & \in \mathit{Persistent variable} \quad \text{Variable} \cap \text{Persistent variable} = \emptyset
\end{align*}
$$

A persistent variable can only appear as the first argument of \text{pread} or \text{pwrite}. Thus, a persistent variable is essentially a label, rather than a first-class value.

3.2 Binding-Time Annotation

Binding times are static, $S$, and dynamic, $D$, where $S \sqsubseteq D$. The language of binding-time annotated declarations $\hat{d}$, statements $\hat{s}$, and expressions $\hat{e}$ is defined as follows:

$$
\begin{align*}
\mathit{b} & \in \mathit{Binding time} = \{ S, D \} \\
\hat{d} & \in \mathit{BT-Declaration} ::= \text{int } x^b | \text{persistent int } p^b \\
\hat{s} & \in \mathit{BT-Statement} ::= x = e^b | \text{pwrite}(p^b, e'^b) | \text{if}(e^b) \hat{s}_1 \text{ else } \hat{s}_2 \\\n& \hspace{1cm} | \text{while}(e^b) \hat{s} \{\hat{s}_1; \ldots; \hat{s}_n\} \\
\hat{e} & \in \mathit{BT-Expression} ::= c | x | \mathit{e}_1^{b_1} \text{ op } \mathit{e}_2^{b_2} | \text{pread}(p^b)
\end{align*}
$$

Figure 1 presents inference rules that specify binding-time annotations for a program, based on an environment $\Gamma$ mapping variables and persistent variables to binding times. In the annotation of a program, $\Gamma(d)$ represents the binding time associated by $\Gamma$ to the variable declared by $d$. The annotation of a statement $s$ is described by $\Gamma, b_s \vdash_s s : \hat{s}$, where the binding time $b_s$ is the least upper bound of the binding-times of the enclosing conditional and loop tests. The annotation of an expression $e$ is described by $\Gamma \vdash_e e : \hat{e}^b$. Annotations can be automatically inferred using standard techniques [11].

The rules of Figure 1 treat statements and expressions as follows. The annotation of an assignment statement is determined by the binding time of the
Programs:
\[ \Gamma, S \vdash s : \hat{s} \]
\[ \Gamma \vdash_p d_1 \ldots d_n, s : d_1^{(a_1)} \ldots d_n^{(a_1)} \hat{s} \]

Statements:
\[ \Gamma \vdash e : e' \quad \Gamma \vDash_b \Gamma \vdash e : e' \]
\[ \Gamma \vdash e : e' \quad \Gamma \vDash_b \Gamma \vdash e : e' \]
\[ \Gamma \vdash e : e' \quad \Gamma \vDash_b \Gamma \vdash e : e' \quad \Gamma \vDash_b \Gamma \vdash e : e' \]
\[ \Gamma \vdash e : e' \quad \Gamma \vDash_b \Gamma \vdash e : e' \]

Expressions:
\[ \Gamma \vdash e : e' \]
\[ \Gamma \vDash_b \Gamma \vdash x : x' \]
\[ \Gamma \vdash e_1 : e_1' \quad \Gamma \vdash e_2 : e_2' \]

Fig. 1. Binding-time analysis

assigned variable, which must be greater than or equal to the binding time of the right-hand side expression and the binding times of the enclosing conditional and loop tests \((b_c)\). The annotation of a \texttt{pwrite} statement is similarly constrained. In the annotation of a conditional statement, the least upper bound of \(b_c\) and the binding time of the test expression is propagated to the analysis of the branches. The annotation of a loop is similar. The result of a \texttt{pread} expression is always dynamic; the binding time of its argument is obtained from the environment. The treatment of the other constructs is straightforward.

This analysis is flow-insensitive and does not allow static assignments under dynamic conditionals and loops. These restrictions can be removed for ordinary variables by existing techniques [9]. Nevertheless, persistent variables must be flow-insensitive, to ensure that every assignment to a persistent variable occurs before any access. An implementation strategy is to perform the binding-time analysis in two phases. The first phase annotates all expressions except persistent variables, using a flow-sensitive analysis. The second phase annotates each persistent variable with the least upper bound of the binding times of all \texttt{pwrite} statements at which it is assigned. This second phase does not affect the binding times of other terms, because the binding time of a \texttt{pread} expression is dynamic, independent of the binding-time of the associated persistent variable, and a \texttt{pwrite} statement has no return value.

### 3.3 Data Specialization

Data specialization stages the source program into a loader and a reader that communicate \emph{via} a cache, which we represent as an array. We thus extend the language with constructs for manipulating cache elements:
\[ s \in \text{Statement} ::= \ldots \mid \{ \text{Cache} \, x; \, s \} \mid *e_1 = e_2 \]
\[ e \in \text{Expression} ::= \ldots \mid *e \]
\[ x \in \text{Variable} \cup \{ \text{cache}, \text{tmp} \} \]

The statement \( \{ \text{Cache} \, x; \, s \} \) is a block that declares a pointer into the cache. Two such pointers are \text{cache}, which is initialized to an external array, and \text{tmp}, which is used in the translation of dynamic conditionals and loops. The indirect assignment \( *e_1 = e_2 \) and the indirect reference \( *e \) initialize and access cache elements, respectively. In practice, bounds checks and casts into and out of a generic cache-element type are added as needed. We refer to this language as the \textit{target language}, and the sublanguage of Section 3.1 as the \textit{source language}.

Figure 2 presents the transformation rules for data specialization of statements and expressions. The transformation of a statement is described by \( i \vdash_s \hat{s} : \langle l, r, i' \rangle \), where \( i \) is the offset from \text{cache} of the next free cache entry, \( l \) is a statement representing \( \hat{s} \) in the loader, \( r \) is a statement representing \( \hat{s} \) in the reader, and \( i' \) is the offset from \text{cache} of the next free cache entry after executing either \( l \) or \( r \). By keeping track of the cache offset \( i \), we reduce the number of assignments to the cache pointer. The transformation of an expression is described by \( i \vdash_{e} \hat{e}^b : \langle l, v, r, i' \rangle \), where \( i \) and \( i' \) represent cache offsets as for statements, \( l \) is a statement initializing the cache with the values of the static subexpressions of \( \hat{e}^b \), \( v \) is an expression to use in the loader to refer to the static value of \( \hat{e}^b \) if it has one, and \( r \) is an expression representing the value of \( \hat{e}^b \) for use in the reader. In the definition of the transformation, \( e \) is the result of removing all binding-time annotations from the annotated expression \( \hat{e}^b \).

The transformation treats statements and expressions as follows. We use cache entries to implement static persistent variables. Thus, \texttt{pwrite}(p^S, \hat{e}^b) is translated into an indirect assignment to the entry allocated to \( p \). This assignment is placed in the loader. Similarly, \texttt{pread}(p^S)^D is translated into an indirect reference to the corresponding entry. This reference is placed in the reader. References and assignments to a dynamic persistent variable are placed in the reader. The treatment of the remaining constructs is standard [5,14]. We include the translation of static conditionals here to give a flavor of the adjustments to the cache pointer needed to implement branching control flow constructs. The complete treatment of conditionals and while loops is presented in Appendix A.

We conclude with \( \text{ds}[p]_\Gamma \), the transformation of a program \( p \) with respect to a binding-time environment \( \Gamma \). Let \( d_1^p \ldots d_n^p \hat{s} \) be the result of annotating \( p \) with respect to \( \Gamma \). Suppose that the first \( m \) declarations of \( p \) declare the static persistent variables \( p_1, \ldots, p_m \). If \( m \vdash_s \hat{s} : \langle l, r, i' \rangle \), then \( \text{ds}[p]_\Gamma \) is:

\[
d_{m+1} \ldots d_n
\{
\text{Cache cache, } p_1, \ldots, p_m;
\text{cache} = \text{cache_start}; \ p_1 = \text{cache}; \ldots; p_m = \text{cache} + m - 1;
\ l; \text{cache} = \text{cache_start}; \ r
\}\
\]
The generated program first initializes cache to the beginning of the cache and the static persistent variables to cache entries. Next the loader \( l \) is executed. Finally, the value of cache is reset, and the reader \( r \) is executed.

4 Correctness

We now relate the semantics of the result of data specialization to the semantics of the source program. We begin with the semantics of the target language, which is a superset of the source language. The semantics depends on a store mapping \( \sigma \) for stores \( \sigma \) and \( \sigma' \). The semantics of an expression \( e \) is specified by \( \sigma \vdash_e e : v \), where \( v \) is a store and \( \sigma \) is a store and \( \sigma' \) is a value. We only describe the semantics of the constructs manipulating persistent variables; the other constructs are standard, and are deferred to Appendix 13.

The semantics must ensure that every reference to a persistent variable using \( \text{pread} \) sees the final value to which the variable is assigned using \( \text{pwrite} \). We use two distinct store locations \( \overline{p}^{\text{in}} \) and \( \overline{p}^{\text{out}} \) to represent each persistent variable \( p \). The location \( \overline{p}^{\text{in}} \) holds the final value of \( p \), while the location \( \overline{p}^{\text{out}} \) records
updates to p. Thus, the semantics of \( \text{pread} \) and \( \text{pwrite} \) are as follows, where \textit{undefined} is some value distinct from the value of any expression:

\[
\begin{align*}
\sigma[\text{p}^{\text{in}} \rightarrow v] & \vdash_{s} \text{pread}(p) : v \\
\sigma \vdash^{s} e : v & \quad \sigma \vdash^{s} \text{pwrite}(p, e) : \sigma[p^{\text{out}} \rightarrow v]
\end{align*}
\]

The values stored in \( p^{\text{in}} \) and \( p^{\text{out}} \) are connected by the semantics of a complete program, specified as follows:

**Definition 1.** Let \( p \) be a program \( d_{1} \ldots d_{n} \) declaring the variables \( x_{1}, \ldots, x_{q} \) and the persistent variables \( p_{1}, \ldots, p_{q} \). Let \( \sigma_{0} \) be a state mapping the \( \bar{x}_{i} \) to the initial values of the \( x_{i} \), and the \( \bar{p}_{j}^{\text{out}} \) to “undefined”. Then, the meaning of \( p, [p] \), is the set of stores \( \sigma \), binding only the \( \bar{x}_{i} \), such that for some values \( v_{1}, \ldots, v_{q} \)

\[
\sigma_{0} \cup \{ \bar{p}_{j}^{\text{in}} \mapsto v_{j} \mid 1 \leq j \leq q \} \vdash^{s} \sigma \cup \{ \bar{p}_{j}^{\text{in}}, \bar{p}_{j}^{\text{out}} \mapsto v_{j} \mid 1 \leq j \leq q \}
\]

This definition uses the value \textit{undefined} to ensure that the meaning of a program represents computations in which the value of a persistent variable is only read when it is also defined.

We now show that data specialization preserves the semantics. Data specialization separates a program into the loader, which only manipulates static variables, and the reader, which only manipulates dynamic variables. To relate the semantics of the loader and reader to the semantics of the source program, we first define the operators \text{stat} and \text{dyn}, which separate the input store into static and dynamic components:

**Definition 2.** Let \( \sigma \) be a store and \( \Gamma \) be a binding-time environment. Let \( p^{\text{out}} \) and \( p^{\text{in}} \) be locations that are unique for each persistent variable \( p \), but that may be identical to each other. Then,

1. \( \text{stat}(\sigma, \Gamma) = \{ \bar{x} \mapsto \sigma(\bar{x}) \mid \Gamma(x) = S \} \cup \{ \bar{p} \mapsto p^{\text{out}}, p^{\text{out}} \mapsto \sigma(p^{\text{out}}) \mid \Gamma(p) = S \} \)
2. \( \text{dyn}(\sigma, \Gamma) = \{ \bar{x} \mapsto \sigma(\bar{x}) \mid \Gamma(x) = D \} \cup \{ \bar{p} \mapsto p^{\text{in}}, p^{\text{in}} \mapsto \sigma(p^{\text{in}}) \mid \Gamma(p) = S \} \cup \{ \bar{p}^{\text{in}} \mapsto \sigma(\bar{p}^{\text{in}}), \bar{p}^{\text{out}} \mapsto \sigma(\bar{p}^{\text{out}}) \mid \Gamma(p) = D \} \)

To relate stores \( \text{stat}(\sigma, \Gamma) \) and \( \text{dyn}(\sigma, \Gamma) \) back to \( \sigma \), we must eliminate the intermediate locations \( p^{\text{in}} \) and \( p^{\text{out}} \). We thus define the operator \( \uplus \):

**Definition 3.** For binding-time environment \( \Gamma \) and stores \( \alpha \) and \( \beta \),

\[
\alpha \uplus_{\Gamma} \beta = \{ \bar{x} \mapsto \alpha(\bar{x}) \mid \Gamma(x) = S \} \cup \{ \bar{x} \mapsto \beta(\bar{x}) \mid \Gamma(x) = D \} \\
\cup \{ \bar{p}^{\text{in}} \mapsto \beta(\bar{p}^{\text{in}}), \bar{p}^{\text{out}} \mapsto \alpha(\bar{p}^{\text{out}}) \mid \Gamma(p) = S \} \\
\cup \{ \bar{p}^{\text{in}} \mapsto \beta(\bar{p}^{\text{in}}), \bar{p}^{\text{out}} \mapsto \beta(\bar{p}^{\text{out}}) \mid \Gamma(p) = D \}
\]

The operators \( \text{stat}, \text{dyn}, \) and \( \uplus \) are related by the following lemma:

**Lemma 1.** \( \text{stat}(\sigma, \Gamma) \uplus_{\Gamma} \text{dyn}(\sigma, \Gamma) = \sigma \)

The loader and reader also use some store locations not present in the store used by the source programs, namely the cache pointer, the local \text{tmp} variables, and the cache. For conciseness, we specify the store with respect to which the loader and reader are executed as a sequence \( \alpha, c, \tau, \xi \) of...
1. The bindings ($\alpha$) associated with the source variables.
2. The cache pointer ($c$).
3. A stack ($\tau$) representing the values of the locally declared tmp variables.
4. The cache ($\xi$).

The following theorem shows that given the loader and reader associated with a source statement, execution of the loader followed by resetting of the cache pointer followed by execution of the reader has the same effect on the values of the source variables as execution of the source statement.

**Theorem 1.** For any statement $s$ and store $\sigma$, if $\Gamma, b_c \vdash s : \hat{s}$ and $i \vdash^s_i \hat{s} : \langle l, r, i' \rangle$, and if for some $c, \tau, \xi$, there are $\alpha, \beta, \zeta, \zeta'$ such that

1. $\text{stat}(\sigma, \Gamma), c, \tau, \xi \vdash^s \sigma : \alpha, \zeta, \zeta'$.
2. $\text{dyn}(\sigma, \Gamma), c, \tau, \xi \vdash^s \sigma : \beta, \zeta, \zeta'$.

Then, $\sigma \vdash^s \sigma : \alpha \uplus \beta$.

Because of speculative evaluation of terms under dynamic control (see Appendix A), termination of the source statement does not necessarily imply termination of the loader. Thus, we relate the semantics of the source program to that of the loader and reader using the following theorem, which includes the hypothesis that the loader terminates.

**Theorem 2.** For any statement $s$ and store $\sigma$, if $\Gamma, b_c \vdash s : \hat{s}$ and $i \vdash^s_i \hat{s} : \langle l, r, i' \rangle$, and there are some $\sigma', c, \tau, \xi$ such that

1. $\sigma' \vdash^s \sigma'$.
2. For some store $\sigma''$, $\text{stat}(\sigma, \Gamma), c, \tau, \xi \vdash^s \sigma'' : \sigma''$

Then, there are some $\alpha, \beta, \zeta, \zeta'$, such that

1. $\text{stat}(\sigma, \Gamma), c, \tau, \xi \vdash^s \sigma : \alpha, \zeta, \zeta'$.
2. $\text{dyn}(\sigma, \Gamma), c, \tau, \xi \vdash^s \sigma : \beta, \zeta, \zeta'$.
3. $\sigma' = \alpha \uplus \beta$

Both theorems are proved by induction on the height of the derivation of $\Gamma, b_c \vdash s : \hat{s}$. These theorems imply the following, which shows that the store resulting from execution of the source statement and the store resulting from execution of the result of data specialization agree on the variables $x$, which determine the semantics of a program as specified by Definition 1.

**Corollary 1.** For any program $p \equiv d_1 \ldots d_n s$, binding-time environment $\Gamma$, and store $\sigma$, if $\text{ds}[p]_{\Gamma} \equiv d'_1 \ldots d'_m s_{ds}, \sigma \vdash^s s : \sigma'$, and $\sigma \vdash^s s_{ds} : \sigma'_{ds}$, then for all variables $x$ of $p$, $\sigma'(x) = \sigma'_{ds}(x)$.

The corollary holds because $s_{ds}$ initially sets up a store that is compatible with the store assumed by the above theorems and resets the cache pointer between the execution of the loader and the reader.
5 Circularity and Attribute Grammars

The efficient implementation of circular specifications has been studied extensively in the attribute-grammar community [13, 18, 19, 20, 21]. As is the case for offline partial evaluation, these approaches begin with a dependency analysis, the results of which then guide code generation. Nevertheless, the partial-evaluation and attribute-grammar-based approaches differ in their starting point and in the quantity of information collected by the analyses. Our approach also differs from attribute-grammar-based approaches in that our source language is imperative.

The starting point of an attribute-grammar-based approach is an attribute grammar describing the input of the program to be generated. An analysis determines the dependencies between the attributes, and uses this information to construct a series of schedules, known as a visit sequences, of the attribute computations such that each computation depends only on previously-computed attributes. Each element of a visit sequence can be implemented as a function whose argument is the component of the input for which the corresponding attribute is to be computed. The resulting implementation amounts to a series of recursive traversals of the input structure [13]. Specific techniques have been devised to transmit intermediate values (“bindings” [18]) that are not part of the input from one phase to the next, and to create specialized input structures (“visit trees” [19, 21]) that eliminate the need to maintain portions of the input that are not needed by subsequent phases.

In contrast, the starting point of data specialization is the program itself; data specialization is independent of the structure of the input data. The construction of both glue and visit trees is subsumed by the basic mechanism of data specialization: the caching of the value of every static expression that occurs in a dynamic context. The cache can, nevertheless, be viewed as implementing a specialized visit tree. The cached values of static conditional tests correspond to sum-type tags, while the values cached for the chosen branch of a static conditional correspond to the components of the sum element. In our implementation, this tree structure is flattened; pointers from one cache entry to another are only introduced to implement speculative evaluation of dynamic control constructs (See Appendix A).

The binding-time analysis used by data specialization is significantly less informative than the analysis used in the implementation of attribute grammars. We have seen that binding-time analysis simply classifies each expression as static or dynamic, according to the classification of the terms that it depends on. This strategy implies that a persistent variable that depends on the value of another instance of itself is considered dynamic. The attribute-grammar analysis collects complete information about the relationships between attributes. This extra precision allows fewer dependencies to be considered recursive. The impact of replacing the binding-time analysis of data specialization by the more informative attribute-grammar analysis is a promising area for future research.

Finally, we have presented an implementation of circularity in the context of an imperative language, whereas attribute grammars use a declarative notation, with some similarity to a lazy functional language [10]. Because imperative lan-
guages are flow sensitive, we have to indicate explicitly which variables should be considered persistent, and thus take on the final value to which they are assigned, rather than the most recent value. In contrast, in a flow-insensitive language, such as a declarative or functional language, a variable has only one value, which is its final value. The addition of circularity to such a language simply allows the value to be specified after the point of reference, rather than requiring that it be specified before, and no keyword is required.

6 Examples

We now present some examples: the translation of the imperative definition of repmin, a use of our approach in the implementation of run-time specialization in Tempo, and two examples from the literature on circular programs.

6.1 Repmin

Figure 3 shows the result of applying data specialization to the imperative implementation of repmin presented in Section 1. The loader, comprised of rmldr and repminldr, accumulates the minimum value at any Tip. Once the complete tree has been analyzed, this value is stored in the cache location assigned to the persistent variable minval. The reader, comprised of rmrdr and repminrdr, uses this value to reconstruct the tree.

In the implementation of Figure 3, calls to the primitives mkTip and mkFork are considered to be dynamic, and thus placed in the reader. It is also possible to consider these calls to be static, in which case the output structure is built in the loader. Following this strategy, the reader recursively visits the tips, instantiating the value of each to be the minimum value. When part of the output structure does not depend on the values of persistent variables, this strategy implies that the binding-time analysis considers its construction to be completely static, which can reduce the amount of data stored in the cache.

6.2 Inlining

This work was motivated by the problem of optimizing run-time specialization in Tempo [15]. An important optimization is the inlining of specialized functions. Inlining is performed during the execution of a dedicated specializer (generating extension [11]) written in C, and is thus most naturally implemented in C as well. We have found that the use of persistent variables facilitates the implementation of various inlining strategies, by requiring only local changes that do not affect the overall implementation of run-time specialization.

To achieve good performance, the size of a specialized function should not exceed the size of the instruction cache or the distance expressible by a relative

---

2 The code produced by Tempo has been slightly modified for readability. Among these simplifications, we exploit the fact that only integers are stored in the cache to eliminate casts to and from a generic cache type.
void rm_ldr(Tree *t, int *cache) {
    Ans a;
    int *minval_ptr = cache++;
    cache =
    repmin_ldr(t, &a, cache);
    *minval_ptr = a.mn;
}

int *repmin_ldr(Tree *t, Ans *a,
                int *cache) {
    Ans a1, a2;
    *cache = (t->type == Fork);
    if (*cache++) {
        cache =
        repmin_ldr(t->left, &a1, cache);
        cache =
        repmin_ldr(t->right, &a2, cache);
        a->mn = min(a1.mn, a2.mn);
    }
    else a->mn = t->tipval;
    return cache;
}

Tree *rm_rdr(int *cache) {
    Ans a;
    int minval = *cache++;
    cache =
    repmin_rdr(minval, &a, cache);
    return a.tree;
}

int *repmin_rdr(int m, Ans *a,
                int *cache) {
    Ans a1, a2;
    if (*cache++) {
        cache =
        repmin_rdr(m, &a1, cache);
        cache =
        repmin_rdr(m, &a2, cache);
        a->tree =
        mkFork(a1.tree, a2.tree);
    }
    else a->tree = mkTip(m);
    return cache;
}

Tree *rm(Tree *t) {
    int *cache = mkCache();
    rm_ldr(t, cache);
    return rm_rdr(cache);
}

Fig. 3. Data specialization of repmin

branch instruction. One approach is to constrain inlining based on the number of instructions already generated for the current function. A more precise approach is to constrain inlining based on the size of the complete specialized function. To implement these strategies, we use data specialization and persistent variables to separate the implementation into a pass that analyzes the size, followed by a pass that performs the inlining and code generation.

Inlining Based on Current Function Size: The heart of the implementation is the function do_call, shown with binding-time annotations in Figure 4. The arguments to do_call are the number of instructions already generated for the caller (caller_size), the name of the callee (callee), and the buffer into which code for the caller is generated (caller_output). The treatment of a call proceeds in three steps. First, callee_output is initialized to the address at which to generate code for the specialized callee. If the specialized callee is to be inlined, callee_output is set to the current position in the caller’s buffer, as indicated
extern Code *the_program[];
extern int threshold;

int do_call(int caller_size, int callee, Code *caller_output) {
    int inlined, callee_size;
persistent int inlinedp, callee_sizep;
    Code *callee_output;

    /* select the output buffer based on whether the call is inlined */
    if (pread(inlinedp))
        callee_output = caller_output;
    else
        callee_output = mkFun(pread(callee_sizep));

    /* specialize the callee */
    callee_size = spec(the_program[ callee ], callee_output);

    /* initializations based on whether the call is inlined */
    inlined = (callee_size + caller_size <= threshold);
pwrite(inlinedp,inlined);
    if (inlined)
        /* return the number of instructions added to the caller */
        return callee_size;
    else {
        /* end the callee */
        *( callee_output + callee_size ) = RETURN;

        /* record the callee’s size */
pwrite( callee_sizep, callee_size+1);
        /* add a call instruction to the caller */
        *output = mkCall( get_name( callee_output ) );

        /* return the number of instructions added to the caller */
        return 1;
    }
}

Fig. 4. The do_call function used in the implementation of inlining based on current function size. Dynamic constructs are underlined.
Implementing Circularity Using Partial Evaluation

should be inlined. The persistent variables are then initialized accordingly, and other initializations are performed as indicated by the comments in Figure 4. The return value is the number of instructions added to the caller.

The loader produced by data specialization computes the size of each generated function and determines whether it should be inlined. The reader then uses this information to allocate the output buffer and perform code generation.

**Controlling Inlining Based on Maximum Possible Function Size:** The previous approach takes into account only the number of instructions generated for the caller so far. A more accurate approach is to consider the caller’s total size. For this purpose we add a new persistent variable `local_sizep` recording the total size of the current function before inlining.

The use of the dynamic value of the persistent variable `local_sizep` to determine whether to inline implies that the value of the persistent variable `inlinedp` depends on dynamic information. Thus, we must iterate data specialization, producing a three-phase implementation. The first phase calculates the number of instructions generated for each source function, if no inlining is performed. The second phase decides whether to inline each call, based on the sum of the number of instructions in the specialized caller and the number of instructions added by inlining all selected calls. The third phase performs the actual code generation.

### 6.3 Other Circular Programs

We now consider several examples from the literature on attribute-grammar-based and lazy implementations of circular programs. These examples illustrate the limitations of an approach based on binding-time analysis.

The Block language has been used by Saraiva et al. to illustrate numerous strategies for generating efficient implementations of attribute grammars [19, 20, 22]. BLOCK is a block-structured language in which the scope of a variable declaration is any nested block, including nested blocks to the left of the declaration. The language thus generalizes the common use of forward references to top-level functions, extending this facility to local variables. The scoping rule is illustrated by the following program, in which braces delimit a nested block, “int x” is a declaration of x, and “x” is a use of x:

```
{ { x } int x }
```

We focus on one of the compilation problems that has been studied for BLOCK, that of translating BLOCK code to a stack-based language [22]. Circularity arises when the compiler needs to determine the stack offset of a variable that occurs before its declaration has been processed.

Saraiva et al. present two implementations of such a compiler: a lazy implementation and a strict implementation generated from an attribute grammar [22]. Both implementations represent the environment as two variables, which we refer to as the local environment and the complete environment. The local environment contains all of the declarations for the enclosing blocks, but only
the variables declared to the left of the current position in the current block. As new declarations are encountered, they are recorded in this environment. The complete environment contains all of the declarations that should be visible at the current point, including those that occur to the right of the current position. This environment is used for code generation. In the compilation of a block, the environments are connected using a circular reference: the input value of the complete environment is the local environment that results from processing the block.

This implementation can be directly translated into our language by using a persistent variable to implement the complete environment. If we follow this strategy, however, our approach is unable to eliminate the circularity. Because the initial value of the local environment is the dynamic value of the persistent variable representing the complete environment of the enclosing block, the local environment is always dynamic, and cannot be used to initialize the complete environment of the nested block to a static value. The strict implementation of Saraiva et al. resolves this dependency by a strategy analogous to calling the loader for the treatment of a block from the reader, rather than from the loader of the context. Within the reader, the complete environment for the surrounding context has been determined. It is, however, not clear how to infer the need for this implementation strategy using binding-time analysis.

By slightly reorganizing the structure of the environment, we can implement the BLOCK compiler using persistent variables and data specialization. The local environment of a nested block extends the enclosing block’s complete environment, but does not otherwise depend on its contents. We thus replace the flat environment generated by Saraiva et al.’s implementation by an environment constructed of frames, such that an empty, and thus static, frame is allocated on entry to each block. The loader adds the declarations made by the block to this frame, which is then stored in a persistent variable at the end of the block. When the reader enters the block, it extends the complete environment with this new frame, producing an environment suitable for code generation. This program structure is also used in the second inlining example. The reorganization corresponds roughly to reformulating a computation \( f(d) \) into \( d \oplus f(s) \), where \( s \) is some static initial value and \( \oplus \) is some operation, thus permitting the computation of \( f \) to be considered static.

A related example is Karczmarczuk’s use of circularity to concisely implement complex mathematical operations in a lazy functional language \[\text{LR}]. Like the compiler of the BLOCK language, Karczmarczuk’s implementation has the property that a circular value from an enclosing computation is used in performing a subcomputation. Here, however, the value from the enclosing computation is used eagerly, and it is not clear how to perform a rewriting of the form of the conversion of \( f(d) \) into \( d \oplus f(s) \).
7 Related Work

The most closely related work is the automatic efficient implementation of an attribute-grammar specification, which has been discussed in Section 5. Here, we review the history of data specialization and of multiple levels of specialization.

*Data specialization:* Automatic data specialization was initially developed by Barzdins and Bulyonkov [2], and described and extended by Malmkjær [16]. These approaches are more complex than ours. In particular, they use memoization in the construction of the data specialization cache. Knoblock and Ruf implement data specialization for a subset of C and investigate its use in an interactive graphics application [14]. Chirokoff et al. compare the benefits of program and data specialization, and propose combining these techniques [5]. Our implementation of data specialization in Tempo builds on that of Chirokoff.

*Incremental specialization:* We have proposed to iterate binding-time analysis and data specialization to resolve dependencies among a hierarchy of persistent variables. Marlet, Consel, and Boinot similarly iterate the specialization process to achieve incremental run-time program specialization [17]. Alternatively, Glück and Jørgensen define a binding-time analysis and program specializer that treat multiple levels at once [7]. Their analysis should be applicable to our approach.

8 Conclusions and Future Work

In this paper, we have shown how circular programs can be implemented using a minor extension of standard partial evaluation techniques. Previously developed techniques to generate optimized implementations of circular specifications are naturally achieved by the basic strategy of caching the values of static expressions that occur in a dynamic context. We have found the use of persistent variables crucial in experimenting with a variety of optimization strategies for run-time specialization in Tempo. Because the introduced code is localized, and a staged program is generated automatically, variants can be implemented robustly and rapidly.

In future work, we plan to allow persistent variables as first-class values. Given the set of analyses already performed by Tempo [8], this extension should be straightforward. We also plan to investigate whether the information collected by the analysis used for attribute grammars can be useful in the context of partial evaluation. We hope that the work presented here will lead to further exchange of techniques between the attribute-grammar and partial-evaluation communities.

**Acknowledgements:** This work has greatly benefitted from the encouragement and feedback of many people, including Olivier Danvy, Andrzej Filinski, Bernd Grobauer, Gilles Muller, Renaud Marlet, Ulrik Schultz, and the PADO-II reviewers.
References


A Data Specialization of Branching Statements

The data specialization rules for conditionals and while loops are shown in Figure 5. The principal problem here is to maintain the cache pointer. For static control constructs, all possible control paths must set the cache pointer such that a single constant offset $i$ can be used after the control construct. The speculative evaluation performed for dynamic control constructs implies that cache entries are initialized in the loader that correspond to code that is not executed in the reader. Thus, the cache itself has to record which cache entries to skip, according to the control path chosen in the reader. While speculative evaluation is not essential, it has been found useful in practice [11].

The specialization rules in Figure 5 create a cache entry for the value of the test of each static conditional and for the value of the test performed on each static while loop iteration. A more efficient approach is to collapse nested static conditionals into a *switch* statement and to replace the recording of the values of while loop tests by the recording of the number of loop iterations. Both optimizations have been implemented in Tempo.

B Semantics

The complete semantics of statements and expressions is shown in Figure 6.
\[ i \vdash^e_d e^S : \langle l, v, r, i' \rangle \quad i' \vdash^s_d \hat{s}_1 : \langle l_1, r_1, i_1 \rangle \quad i' \vdash^s_d \hat{s}_2 : \langle l_2, r_2, i_2 \rangle \]

\[ i \vdash^e_d e^D : \langle l, v, r, i' \rangle \quad i' + 1 \vdash^s_d \hat{s}_1 : \langle l_1, r_1, i_1 \rangle \quad i_1 + 1 \vdash^s_d \hat{s}_2 : \langle l_2, r_2, i_2 \rangle \]

\[ i \vdash^e_d e^S : \langle l, v, r, i' \rangle \quad i' \vdash^s_d \hat{s} : \langle l_s, r_s, i_s \rangle \]

\[ i \vdash^e_d e^D : \langle l, v, r, i' \rangle \quad i' + 1 \vdash^s_d \hat{s} : \langle l_s, r_s, i_s \rangle \]

---

**Fig. 5.** Data specialization of branching statements

**Statements:**

\[
\begin{align*}
\sigma &\vdash^e_S e : v \\
\sigma &\vdash^e_S x = c : \sigma[x \mapsto v] \\
\sigma &\vdash^e_S e_1 : \ell \\
\sigma &\vdash^e_S \ast e_2 : \sigma[\ell \mapsto v] \\
\sigma &\vdash^e_S p \cdot \text{write}(p, e) : \sigma[p^\text{out} \mapsto v] \\
\sigma &\vdash^e_S \text{if} (e) s_1 : \sigma' \\
&\vdash^s_S \text{if} (e) s_1 \text{else } s_2 : \sigma' \\
\sigma &\vdash^e_S \text{while } (e) s : \sigma'' \\
&\vdash^s_S \text{while } (e) s : \sigma'' \\
\sigma &\vdash^e_S \text{if } (e) s_1 \text{else } s_2 : \sigma' \\
\sigma &\vdash^e_S \text{while } (e) s : \sigma'' \\
\sigma &\vdash^e_S s_1 : \sigma_1 \\
&\vdots \\
\sigma &\vdash^e_S s_n : \sigma_n \\
\sigma &\vdash^s_S \{s_1 ; \ldots ; s_n\} : \sigma_n \\
\sigma &\vdash^e_S x \mapsto \text{undefined} \\
&\vdash^s_S s : \sigma' [x \mapsto v] \\
\sigma &\vdash^s_S \{\text{Cache } x ; s\} : \sigma' \\
\end{align*}
\]

**Expressions:**

\[
\begin{align*}
\sigma &\vdash^e_S c : c \\
\sigma &\vdash^e_S x \mapsto v \vdash^e_S x : v \\
\sigma &\vdash^e_S e_1 : v_1 \\
\sigma &\vdash^e_S e_2 : v_2 \\
\sigma &\vdash^e_S e_1 \cdot p \cdot e_2 : v_1 \cdot p \cdot v_2 \\
\sigma &\vdash^e_S e_2 : v \\
\sigma &\vdash^e_S e_1 \cdot p \cdot e_2 : v_1 \cdot p \cdot v_2 \\
\sigma &\vdash^e_S e_2 : v \\
\sigma &\vdash^e_S e_1 \cdot p \cdot e_2 : v_1 \cdot p \cdot v_2 \\
\end{align*}
\]

**Fig. 6.** Semantics of statements and expressions
Combining Forward and Backward Analyses of Temporal Properties

Damien Massé
LIX, École Polytechnique, Palaiseau, France, masse@lix.polytechnique.fr, http://www.lix.polytechnique.fr/~masse/

Abstract. In this paper we extend the well-known combination of forward and backward static analyses in abstract interpretation for the verification of complex temporal properties for transition systems. First, we show that this combination, whose results are often better than those obtained by using both analyses separately, can be used to check simple temporal properties with just one fixpoint. Then we extend this result to more complex temporal properties, including a superset of CTL in the case of non-game properties, and a superset of ATL in the case of game properties.

1 Introduction

Abstract interpretation [4, 7, 9] is a formal method for inferring general properties of a program. When the program is described as a transition system, two kinds of analyses can be done: backward analysis and forward analysis. Forward analysis simulates program computations, whereas backward analysis simulates reverse computations. Both analyses can be combined to obtain much better results, since each analysis may reduce the loss of precision introduced by the other [4]. However, only restricted kinds of properties expressed in the μ-calculus as intersections of properties in the form of \( \nu X. (p \land \bigtriangleup X) \) and \( \mu X. (p \lor \bigtriangledown X) \) are used, including for user-provided assertions [1].

More complex temporal properties (such as CTL and ATL [1]) are commonly checked in the model checking approach [3]. In that case, abstractions are commonly used to solve state explosion problems. Anyway, model-checking tools use either backward or forward analyses [12], but usually do not combine them, since one analysis is enough for finite concrete systems, and reversible temporal logics [10] are not used for specifications (forward logics, that use only predecessor operators, are used instead [4]).

In this paper, we show that forward analysis can still be combined with backward analysis in many model-checking temporal specifications. We will study

1 In fact, the implemented tool [2] can be used to prove the negation of these properties, not these properties.

2 It is shown in [10] that state-based abstractions are not complete when checking reversible temporal specifications.
non-game properties (a subset of $\mu$-calculus formulas), and game properties (a subset of $A\mu$ formulas [1]). This combination can lead to better results, especially when using widening and narrowing techniques [4] to deal with infinite abstract lattices.

2 Standard Combination of Backward and Forward Analysis

The combination of backward and forward analyses was originally introduced in Cousot’s thesis [4] in order to approximate the intersection of backward and forward transition system collecting semantics. It is widely known and used in abstract interpretation, since it enables to combine information given by abstract backward and forward operators, and thus to reduce the effects of the loss of information due to the abstraction.

In this section we recall the results that justify the correctness of this combination.

2.1 Combination of Fixpoints

**Lemma 1.** Let $P^\flat((\bot^\flat, \top^\flat, \sqcap^\flat, \sqcup^\flat))$ and $P^\sharp((\bot^\sharp, \top^\sharp, \sqcap^\sharp, \sqcup^\sharp))$ be complete lattices with a Galois connection $\gamma : P^\flat \rightarrow P^\sharp$, $\alpha : P^\sharp \rightarrow P^\flat$ be two monotonic functions, and let

$$L^\flat = \text{gfp } \lambda Z. (Z \sqcap \top^\flat F^\flat(Z) \sqcap \top^\sharp B^\sharp(Z))$$

If $F^\sharp \in P^\sharp \rightarrow P^\sharp$ and $B^\sharp \in P^\sharp \rightarrow P^\sharp$ are monotonic and satisfy $\alpha \circ F^\flat \circ \gamma \sqsubseteq F^\sharp$ and $\alpha \circ B^\flat \circ \gamma \sqsubseteq B^\sharp$, then the sequence $(X_n)_{n \in \mathbb{N}}$ defined by $X_0 = \top^\sharp$, $X_{2n+1} = X_{2n} \sqcap F^\sharp(X_{2n})$, and $X_{2n+2} = X_{2n+1} \sqcup B^\sharp(X_{2n+1})$, $\forall n \geq 0$ is such that:

$$\forall n \geq 0, \alpha(L^\flat) \sqsubseteq X_{n+1} \sqsubseteq X_n$$

The optimality of this approach has been proved in [5]: it has been shown that $L^\sharp = \text{gfp } \lambda Z. (Z \sqcap F^\sharp(Z) \sqcap B^\sharp(Z))$ is the greatest lower bound of the set $E^\sharp$ defined inductively as:

- $\top^\sharp \in E^\sharp$
- If $Z$ is in $E^\sharp$ then so are $F^\sharp(Z)$ and $B^\sharp(Z)$.
- If $Z_1$ and $Z_2$ are in $E^\sharp$ then so are $Z_1 \sqcap \sharp Z_2$ and $Z_1 \sqcup \sharp Z_2$.

Therefore, $L^\sharp$ is the best upper approximation of $\alpha(L^\flat)$ that can be obtained using $F^\sharp$ and $B^\sharp$.

2.2 Standard Backward-Forward Combination

The standard backward-forward combination [3] derives from an application of this lemma to the particular case of backward and forward collecting semantics: $F^\flat$, $B^\flat$, $F^\sharp$ and $B^\sharp$ are instantiated as follows:
\[
F^\flat = \lambda Y. \lgfp_1 \lambda X. (Y \uparrow \flat f^\flat(X)) \\
B^\flat = \lambda Y. \lgfp_2 \lambda X. (Y \uparrow \flat b^\flat(X)) \\
F^\sharp = \lambda Y. \lgfp_1 \lambda X. (Y \uparrow \sharp f^\sharp(X)) \\
B^\sharp = \lambda Y. \lgfp_2 \lambda X. (Y \uparrow \sharp b^\sharp(X))
\]

where \(\lgfp\) means either \(\text{lfp}\) or \(\text{gfp}\), and \(f^\flat, b^\flat \in \text{P}^\flat \rightarrow \text{P}^\flat\), \(f^\sharp, b^\sharp \in \text{P}^\sharp \rightarrow \text{P}^\sharp\) are monotonic. When \(\alpha \circ f^\flat \circ \gamma \subseteq f^\sharp\) and \(\alpha \circ b^\flat \circ \gamma \subseteq b^\sharp\), the conditions of Lemma 1 are satisfied \cite{7}.

Now, let \(P^\flat = \wp(\Sigma)\), \(\Sigma\) a set of states\(^3\), and \(\tau \in \wp(\Sigma \times \Sigma)\) a transition relation. As usual, we define \(\text{pre}, \tilde{\text{pre}}, \text{post}, \tilde{\text{post}}\) as:

\[
\text{post}(X) = \{s' \mid \exists s : (s, s') \in \tau \land s \in X\} \\
\tilde{\text{post}}(X) = \{s' \mid \forall s : (s, s') \in \tau \Rightarrow s \in X\} \\
\text{pre}(X) = \{s \mid \exists s' : (s, s') \in \tau \land s' \in X\} \\
\tilde{\text{pre}}(X) = \{s \mid \forall s' : (s, s') \in \tau \Rightarrow s' \in X\}
\]

Given \(\mathcal{T}, \mathcal{F} \subseteq \Sigma\), sets of initial and final states, \(f^\flat = \lambda X. (\mathcal{I} \cup \text{post}(X))\) and \(b^\flat = \lambda X. (\mathcal{F} \cup \text{pre}(X))\) (and \(\lgfp_1 = \text{lgfp}_2 = \text{lfp}\)), we have \cite{4}:

\[
L^\flat = F^\flat(\top^\flat) \uparrow \flat B^\flat(\top^\flat) = \text{lfp} f^\flat \cap \text{lfp} b^\flat \tag{1}
\]

By computing \(\gamma(L^\flat)\), we obtain a good upper approximation of \(F^\flat(\Sigma) \cap B^\flat(\Sigma)\) (at least equal to \(\gamma(F^\sharp(\top^\sharp) \cap B^\sharp(\top^\sharp))\)).

\(B^\flat(\top^\flat)\) is the set of states satisfying the \(\mu\)-calculus formula \(\mu X. (F \lor \Box X)\), where \(F\) is satisfied by \(\mathcal{F}\). Therefore the method is used to analyze reachability (rather unreachability, since we compute a superset of reachable states, or a subset of unreachable states) properties. Equation \(\text{(1)}\) holds with the \(\mu\)-calculus formula \(\nu X. (F \land \Box X)\) too, and this result allows the analysis of termination properties.

For other formulas (like \(\mu X. (F \lor \Box X)\)), equation \(\text{(1)}\) does not hold in general: a state may satisfy the backward property and be reachable from an initial state which does not satisfy the backward property (an example is given in Figure \(\text{I}\)).

In the next section, we will prove that under certain conditions we can still use the combination for formulas with a single fixpoint.

### 3 Extension with a Single Fixpoint

As we want to deal with \(\mu\)-calculus formulas in this section, we assume that \(f^\flat = \lambda X. (\mathcal{I} \cup \text{post}(X))\) and that \(\lgfp_1 = \text{lfp}\). Computing (or approximating) \(\alpha(L^\flat)\) is useless if we do not know \(L^\flat\). Equation \(\text{(1)}\) must hold in order to approximate the sets of descendant states of \(\mathcal{I}\) which satisfy \(B^\flat(\top^\flat)\). Anyway, if we want to

\(\text{So } \top^\flat = \Sigma, \bot^\flat = \emptyset, \cap^\flat = \cap, \cup^\flat = \cup.\)

\(\text{4 We may over-approximate the greatest fixpoint with a narrowing }\text{[7]}.\)
Fig. 1. Example where \( L^b \neq F^b(\top) \cap B^b(\top) \). Here, \( b^b = \lambda X.(F \cup \overline{\text{pre}}(X)) \) and \( f^b = \lambda X.(\mathcal{I} \cup \text{post}(X)) \). \( L^b \) is then the set of states belonging to a trace of states which satisfy the backward property.

check a temporal formula, we just need to know the set of initial states which satisfy \( B^b(\top) \), that is \( \mathcal{I} \cap B^b(\top) \). Thus the combination is useful if the equality:

\[
\mathcal{I} \cap B^b(\Sigma) = \mathcal{I} \cap L^b
\]

is satisfied\(^5\).

**Lemma 2.** Assuming that \( f^b = \lambda X.(\mathcal{I} \cup \text{post}(X)) \), if \( L^b = F^b(B^b(\Sigma)) \), then \( \mathcal{I} \cap B^b(\Sigma) = \mathcal{I} \cap L^b \).

*Proof.* As \( L^b = F^b(B^b(\Sigma)) = \text{lfp} \lambda X.(B^b(\Sigma) \cap (f^b(X))) \), it is clear that \( L^b \subseteq B^b(\Sigma) \). Moreover, the first iteration of the least fixpoint is \( B^b(\Sigma) \cap f^b(\emptyset) \), which is equal to \( B^b(\Sigma) \cap \mathcal{I} \). So we have \( B^b(\Sigma) \cap \mathcal{I} \subseteq L^b \), which proves the equality.

With \( B^b = \lambda Y.\text{lgfp}_2 \lambda X.(Y \cap b^b(X)) \), we have the following lemma:

**Lemma 3.** If \( \forall(X,Y) \in \wp(\Sigma)^2, Y \subseteq b^b(X) \iff Y \subseteq b^b( X \cap \text{post}(Y)) \), then the hypothesis of Lemma 2 holds. Thus, equation (2) holds.

*Proof.* We note that the hypothesis implies:

\[
\forall(X,Y) \in \wp(\Sigma)^2, Y \subseteq b^b(X) \iff Y \subseteq b^b( X \cap f^b(Y))
\]

\(^5\) And this equality is satisfied when equation (1I) holds.
We want to prove that $L^b = F^b(B^b(\Sigma))$. Left inclusion is the consequence of the optimality of $L^b$:

$$
L^b = \text{gfp } \lambda Z.(Z \cap F^b(Z) \cap B^b(Z)) \\
\subseteq F^b(\langle F^b(\Sigma) \cap B^b(\Sigma) \rangle \cap B^b(F^b(\Sigma) \cap F^b(\Sigma))) \\
\subseteq F^b(B^b(\Sigma))
$$

Thus, to prove the equality, we need to check that $F^b(B^b(\Sigma))$ is a fixpoint of $\lambda Z.(Z \cap F^b(Z) \cap B^b(Z))$, that is, to prove that $F^b(B^b(\Sigma)) \subseteq F^b(F^b(B^b(\Sigma)))$ and $F^b(B^b(\Sigma)) \subseteq B^b(F^b(B^b(\Sigma)))$. The former inequality is true because $F^b \circ F^b = F^b$. To prove the latter, we define $\Omega = F^b(B^b(\Sigma))$ and $\Omega' = B^b(F^b(B^b(\Sigma)))$.

We distinguish two cases:

- if $\text{lgfp}_2 = \text{lfp}$, let $X_n, n \geq 0$ be the (transfinite) iteration sequence starting from $\emptyset$ for $b^\flat$. We will prove that $\Omega \cap X_n \subseteq \Omega'$, for all $n \geq 0$. This is true if $n = 0$, because $X_0 = \emptyset$. If $n$ is a successor ordinal, and if the inequality holds for $n - 1$, we have $\Omega \cap X_n \subseteq b^\flat(X_{n-1})$. Using the hypothesis of the lemma, we obtain $\Omega \cap X_n \subseteq b^\flat(X_{n-1} \cap f^\flat(\Omega \cap X_n))$. By monotony, $f^\flat(\Omega \cap X_n) \subseteq f^\flat(\Omega)$. So, since $X_{n-1} \subseteq B^\flat(\Sigma)$:

$$
X_{n-1} \cap f^\flat(\Omega \cap X_n) \subseteq X_{n-1} \cap B^\flat(\Sigma) \cap f^\flat(\Omega) \\
\subseteq X_{n-1} \cap \Omega \\
\subseteq \Omega'
$$

Hence $\Omega \cap X_n$ is included in $\Omega \cap b^\flat(\Omega')$, which is equal to $\Omega'$ by definition of $\Omega'$.

When $n$ is a limit ordinal ($b^\flat$ may be not continuous), if $\Omega \cap X_{n'} \subseteq \Omega'$ for all $n' < n$, then $\Omega \cap X_n = \Omega \cap \bigcup_{n' < n} X_{n'} \subseteq \Omega'$. By transfinite induction, $\Omega \cap X_n \subseteq \Omega'$ for all $n$. As the upper bound of $(X_n)$ is $B^\flat(\Sigma)$, which includes $\Omega$, we have $\Omega \subseteq \Omega'$.

- if $\text{lgfp}_2 = \text{gfp}$, let $X_n, n \geq 0$ be the (transfinite) iteration sequence starting from $\Sigma$ for $\lambda X.(\Omega \cap b^\flat)$. The limit of $X_n$ is $\Omega'$. $X_1 = \Omega \cap b^\flat(\Sigma) = \Omega$, since $\Omega \subseteq B^\flat(\Sigma) \subseteq b^\flat(\Sigma)$. Moreover, since $B^\flat(\Sigma) = b^\flat(B^\flat(\Sigma))$, $\Omega \subseteq b^\flat(B^\flat(\Sigma))$, so $\Omega \subseteq b^\flat(B^\flat(\Sigma) \cap f^\flat(\Omega))$. As $B^\flat(\Sigma) \cap f^\flat(\Omega) = \Omega$, we have $\Omega \subseteq b^\flat(\Omega)$, and $X_2 = \Omega$. Thus $X_n = \Omega$ for all $n \geq 1$, and $\Omega' = \Omega$.

**Application:** With $b^\flat = \lambda X.(A \cup (B \cap \text{pre}(X)) \cup (C \cap \text{pre}(X)))$.
If $Y \subseteq b^\flat(X)$, then, $\forall y \in Y$:

- if $y \in A$, then $y \in b^\flat(X \cap \text{post}(Y))$.
- if $y \in B \cap \text{pre}(X)$, then $\exists x \in X$ such that $\langle y, x \rangle \in \tau$. Therefore, since $y \in Y$, $x \in \text{post}(Y)$, and $y \in B \cap \text{pre}(X \cup \text{post}(Y))$.
  Thus $y \in b^\flat(X \cap \text{post}(Y))$.
- if $y \in C \cap \text{pre}(X)$, then $\forall x \in \Sigma$, $\langle y, x \rangle \in \tau \Rightarrow x \in X$. As $y \in Y$, $\langle y, x \rangle \in \tau \Rightarrow x \in \text{post}(Y)$, so $\forall x \in \Sigma$, $\langle y, x \rangle \in \tau \Rightarrow x \in X \cap \text{post}(Y)$.
  Thus $y \in C \cap \text{pre}(X \cup \text{post}(Y))$, so $y \in b^\flat(X \cap \text{post}(Y))$. 

Therefore, \( Y \subseteq b^\flat(X) \Rightarrow Y \subseteq b^\flat(X \cap post(Y)) \). The other side of the equivalence is automatic. Thus the hypothesis of Lemma 3 is satisfied, and equation (2) holds.

So we can use the backward-forward combination to enhance the verification of properties in the form of: \( \sigma X.(A \lor (B \land X) \lor (C \land \Box X)) \) with \( \sigma \in \{\mu, \nu\} \). These properties are interesting; they allow to distinguish between different kinds of non-determinism (“controllable” and “uncontrollable” non-determinism). We are not far from game properties, as we will see in section 5.

Unfortunately, the extension of this technique to the whole \( \mu \)-calculus does not work: for example, a formula like \( \mu X.(F \lor \Diamond X) \) leaves “holes” in traces, preventing combination with forward analysis. However, it is possible, from this result, to extend it to a temporal logic expressive at least as C\( \text{TL} \).

### 4 Extension to a Larger Specification Language

In this section, we try to apply the backward-forward combination to the verification of some \( \mu \)-calculus formulas. If \( \varphi \) is a formula, we denote by \([\varphi]\) the set of states (in \( \Sigma \)) satisfying \( \varphi \).

The formulas \( \varphi \) are defined by the grammar:

\[
\varphi ::= p \mid \neg p \mid \varphi_1 \land \varphi_2 \mid \varphi_1 \lor \varphi_2 \mid \Box \varphi \mid \Diamond \varphi \mid \sigma X.(\varphi_1 \lor (\varphi_2 \land X) \lor (\varphi_3 \land \Box X))
\]

with \( \sigma \in \{\mu, \nu\} \). It is worth noting that all these formulas are closed, and the defined temporal logic includes C\( \text{TL} \). Obviously, the logic does not change if we replace \( \sigma X.(\varphi_1 \lor (\varphi_2 \land X) \lor (\varphi_3 \land \Box X)) \) with \( \sigma X.(\varphi_1 \land (\varphi_2 \lor X) \land (\varphi_3 \lor \Box X)) \) in the above grammar.

Our goal is to obtain a “good” upper approximation \( \Omega_{\varphi}(\alpha(I)) \) of \( \alpha(I \cap [\varphi]) \), using backward-forward technique to enhance fixpoint computations. We assume that for all proposition \( p \), we have an upper approximation \([p]^{\sharp}\) of \( \alpha([p])^{\sharp} \) (and an upper approximation \([\neg p]^{\sharp}\) of \( \alpha([\neg p])^{\sharp} \)).

We need abstractions of \( pre, post \) and \( \tilde{pre} \), and we will respectively denote them by \( pre^{\sharp}, post^{\sharp} \) and \( \tilde{pre}^{\sharp} \). Moreover we will denote by \( post^{*} \) and \( post^{* \sharp} \) the functions \( \lambda X.\text{lfp} \lambda Y.(X \cup post(Y)) \) and \( \lambda X.\text{lfp} \lambda Y.(X \cup^{\sharp} post^{\sharp}(Y)) \) respectively. The following inequalities are assumed to be satisfied for all \( X \subseteq \Sigma \):

\[
\begin{align*}
\alpha \circ pre(X) & \subseteq^{\sharp} pre^{\sharp} \circ \alpha(X) \\
\alpha \circ \tilde{pre}(X) & \subseteq^{\sharp} \tilde{pre}^{\sharp} \circ \alpha(X) \\
\alpha \circ post(X) & \subseteq^{\sharp} post^{\sharp} \circ \alpha(X) \\
\alpha \circ post^{*}(X) & \subseteq^{\sharp} post^{* \sharp} \circ \alpha(X)
\end{align*}
\]

\(6\) which may be \( \alpha([p])^{\sharp} \), if it is computable.
To simplify notations we denote by $L^\sharp_{lgfp}(f,g)$, with $\sharp \in \{\flat,\sharp\}$, the expression:

$$\text{gfp}\,\lambda Z.\,(\text{lfp}\,\lambda X.\,(Z \sqcap^\sharp f(X)) \sqcap^\sharp \text{lfp}\,\lambda X.\,(Z \sqcap^\sharp g(X)))$$

$L^\sharp_{lgfp}(f,g)$ is the limit of the decreasing chain $Z_n$ defined by $Z_0 = \top^\sharp$, $Z_{2n+1} = Z_{2n} \sqcap^\sharp \text{lfp}\,\lambda X.\,(X_{2n} \sqcap^\sharp f(X))$ and $Z_{2n+2} = Z_{2n+1} \sqcap^\sharp \text{lfp}\,\lambda X.\,(X_{2n+1} \sqcap^\sharp g(X)).$

If $\varphi$ is a formula, we can now define $\Omega_\varphi \in P^\sharp \to P^\sharp$ as follows:

$$\begin{align*}
\Omega_p(S) &= [p]^\sharp \sqcap^\sharp S \\
\Omega_{\varphi_1 \land \varphi_2}(S) &= \Omega_{\varphi_1}(S) \sqcap^\sharp \Omega_{\varphi_2}(S) \\
\Omega_{\varphi_1 \lor \varphi_2}(S) &= \Omega_{\varphi_1}(S) \sqcup^\sharp \Omega_{\varphi_2}(S) \\
\Omega_{\lnot \varphi}(S) &= \tilde{\text{pre}}^\sharp(\Omega_{\varphi}(\text{post}^\sharp(S))) \\
\Omega_{\varphi \land \sigma X}(S) &= L^\sharp_{lgfp}(\lambda X.(S \sqcup^\sharp \text{post}^\sharp(X)), \lambda Y.(\Omega_{\varphi_1}(\text{post}^\sharp(S)) \sqcup^\sharp \Omega_{\varphi_2}(\text{post}^\sharp(S))) \sqcap^\sharp \tilde{\text{pre}}^\sharp(Y)) \\
\Omega_{\varphi \lor \sigma X}(S) &= L^\sharp_{lgfp}(\lambda X.(S \sqcup^\sharp \text{post}^\sharp(X)), \lambda Y.(\Omega_{\varphi_1}(\text{post}^\sharp(S)) \sqcup^\sharp \Omega_{\varphi_2}(\text{post}^\sharp(S))) \sqcap^\sharp \tilde{\text{pre}}^\sharp(Y))
\end{align*}$$

It is worth noting that even if we replace $\text{post}^\sharp(S)$ by $\top^\sharp$ in the last line, we still have to compute it as the first iteration that leads to $L^\sharp_{lgfp}$. However, this replacement may not change the final result of $\Omega_\varphi$ and make the computation much faster (because the computation of $\Omega_\varphi(\top^\sharp)$ can be simplified). The following theorem is valid with or without the replacement:

**Theorem 1.** For any formula $\varphi$, and $I \subseteq \Sigma$:

$$\alpha(I \cap [\varphi]) \sqsubseteq^\sharp \alpha(I) \sqcap^\sharp \Omega_\varphi(\alpha(I))$$

**Proof.** The proof is by induction on the structure of $\varphi$. By monotony of $\alpha$, it is obvious that $\alpha(I \cap [\varphi]) \sqsubseteq^\sharp \alpha(I)$, so we need to prove that $\alpha(I \cap [\varphi]) \sqsubseteq^\sharp \Omega_\varphi(\alpha(I))$.

If $\varphi = p$, by monotony of $\alpha$:

$$\alpha(I \cap [p]) \sqsubseteq^\sharp \alpha(I) \sqcap^\sharp \alpha([p]) \sqsubseteq^\sharp \alpha(I) \sqcap^\sharp [p]^\sharp \sqsubseteq^\sharp \Omega_p(\alpha(I))$$

If $\varphi = \varphi_1 \land \varphi_2$:

$$\begin{align*}
\alpha(I \cap [\varphi]) &= \alpha(I \cap [\varphi_1] \sqcap I \cap [\varphi_2]) \\
&\sqsubseteq^\sharp \alpha(I \cap [\varphi_1]) \sqcap^\sharp \alpha(I \cap [\varphi_2]) \\
&\sqsubseteq^\sharp \Omega_{\varphi_1}(\alpha(I)) \sqcap^\sharp \Omega_{\varphi_2}(\alpha(I)) \\
&\sqsubseteq^\sharp \Omega_\varphi(\alpha(I))
\end{align*}$$

If $\varphi = \varphi_1 \lor \varphi_2$, the calculus is quite the same, except that we use the fact that $\alpha$ is additive \[8\].
If $\varphi = \Box \varphi_1$, we have $I \subseteq \operatorname{pre}(\operatorname{post}(I))$, so:

$$
\alpha(I \cap [\varphi]) \subseteq^z \alpha(\operatorname{pre}(\operatorname{post}(I) \cap [\varphi_1]))
\subseteq^z \operatorname{pre}^z(\alpha(\operatorname{post}(I) \cap [\varphi_1]))
\subseteq^z \operatorname{pre}^z(\Omega_{\varphi_1}(\alpha \circ \operatorname{post}(I)))
\subseteq^z \operatorname{pre}^z(\Omega_{\varphi_1}(\operatorname{post}^z \circ \alpha(I)))
\subseteq^z \Omega_{\varphi}(\alpha(I))
$$

If $\varphi = \diamond \varphi_1$, using the inequality $I \cap \operatorname{pre}([\varphi_1]) \subseteq \operatorname{pre}(\operatorname{post}(I) \cap [\varphi_1])$, we can do the same calculus.

If $\varphi = \sigma X.((\varphi_1 \lor (\varphi_2 \land \Box X) \lor (\varphi_3 \land \Box X))$, let’s define $h^b = \lambda X.([\varphi_1] \cup [\varphi_2] \cap \operatorname{pre}(X) \cup [\varphi_3] \cap \operatorname{pre}(X))$. Then $[\varphi] = \operatorname{lngf} \lambda X. h^b(X)$.

We will use Lemma 4 with

$$
\begin{align*}
f^b &= \lambda X.(I \cup \operatorname{post}(X)) \\
\varphi^b &= \lambda X.(\alpha(I) \cup \varphi) \\
b^b &= \lambda X.((\operatorname{post}^*(I) \cap ([\varphi_1] \cup [\varphi_2] \cap \operatorname{pre}(X) \cup [\varphi_3] \cap \operatorname{pre}(X))) \\
\Gamma^b &= \lambda X.((\alpha(\operatorname{post}^*(I) \cap [\varphi_1]) \\
\varphi^b &= \lambda X.((\alpha(\operatorname{post}^*(I) \cap [\varphi_2]) \cap \operatorname{pre}^*(X)) \\
\varphi^b &= \lambda X.((\alpha(\operatorname{post}^*(I) \cap [\varphi_3]) \cap \operatorname{pre}^*(X)) \\
\end{align*}
$$

It is clear that $\alpha \circ b^b \circ \gamma \subseteq^z b^\sharp$ and $\alpha \circ f^b \circ \gamma \subseteq^z f^\sharp$ (given the standard properties that $\alpha$ is additive and $\alpha \circ \gamma$ is a lower closure operator [7]). Thus we have $\alpha(L_{\operatorname{lngf}}^b(f^b, b^b)) \subseteq^z L_{\operatorname{lngf}}^\sharp(f^\sharp, b^\sharp)$.

First, we prove that $I \cap L_{\operatorname{lngf}}^b(f^b, b^b) = I \cap [\varphi]$. We have:

$$
L_{\operatorname{lngf}}^b(f^b, h^b) \subseteq \operatorname{lngf} \lambda X.((\operatorname{lfp} \lambda Y. f^b(Y)) \cap h^b(X)) \subseteq [\varphi]
$$

Applying Lemma 9 with $h^b$, we obtain $I \cap [\varphi] = I \cap L_{\operatorname{lngf}}^b(f^b, h^b)$, so $I \cap [\varphi] = I \cap \operatorname{lngf} \lambda X.((\operatorname{lfp} \lambda Y. f^b(Y)) \cap h^b(X)) = I \cap \operatorname{lngf} \lambda X. b^b(X)$. Then, applying Lemma 9 with $b^b$, we obtain: $I \cap \operatorname{lngf} \lambda X. b^b(X) = I \cap L_{\operatorname{lngf}}^b(f^b, b^b)$, proving the equality. So we proved:

$$
\alpha(I \cap [\varphi]) = \alpha(I \cap L_{\operatorname{lngf}}^b(f^b, b^b)) \\
\subseteq^z \alpha(I \cap \varphi) \cap \alpha(L_{\operatorname{lngf}}^b(f^b, b^b)) \\
\subseteq^z \alpha(I \cap \varphi) \cap L_{\operatorname{lngf}}^\sharp(f^\sharp, b^\sharp)
$$

Now we need to check that:

$$
L_{\operatorname{lngf}}^\sharp(f^\sharp, b^\sharp) \subseteq^z \Omega_{\varphi}(\alpha(I))
$$

By induction hypothesis, we see that $\Omega_{\varphi}(\alpha(I)) = L_{\operatorname{lngf}}^\sharp(f^\sharp, b^\sharp)$ with $b^\sharp$ satisfying $b^\sharp \subseteq^z b^\sharp$, which complete the proof.
5 Extension to Alternating-Time Temporal Logic

Many properties on reactive systems are not easily expressible as $\mu$-calculus formulas. This is true for game properties, which can be expressed as alternating time $\mu$-calculus ($A\mu$) formulas \[^1\] or as formulas of weaker game logics $\text{Atl}$ and $\text{Atl}^\ast$. In \[^13\], basic abstract interpretation theory was applied on alternating transition systems, with a model-checking approach of abstraction. As we did with a subset of $\mu$-calculus, we will try to apply forward-backward techniques to $A\mu$.

5.1 Alternating Transition System, Operators

An alternating transition system \[^1\] is a tuple $\langle \Sigma, Q, \Delta, \Pi, \pi \rangle$, with $\Sigma$ a set of players, $Q$ a set of states, $\Delta = \{ \delta_i : Q \rightarrow 2^{2^Q} \mid i \in \Sigma \}$ a set of transition functions, $\Pi$ a set of propositions, and $\pi : \Pi \rightarrow 2^Q$ a function which associates each proposition to a set of states.

When the system is in state $q$, each player $a$ must choose a set $Q_a \in \delta_a(q)$, and the successor of the state $q$ must lie in $\bigcap_{a \in \Sigma} Q_a$ (it is assumed that the intersection is always a singleton, so the transition function is nonblocking and “deterministic”). Thus, if we want an equivalent of the $\text{post}$ operator used in the non-game case, it would be:

$$\text{Post}(\sigma) = \bigcup_{q \in \sigma} \bigcap_{a \in \Sigma} \delta_a(q)$$

As before, we can define $\text{Post}^*(\sigma) = \text{lfp } \lambda X.(\sigma \cup \text{Post}(X))$.

The equivalent of the $\text{pre}$ and $\text{pre}^\ast$ operators are the controllable and uncontrollable predecessor relations, defined in \[^13\]. In general case, with $I \subset \Sigma \setminus \{\emptyset\}$, they are defined as:

$$q \in \text{CPre}_I(\sigma) \text{ iff } \exists (\tau_i \in \delta_i(q))_{i \in I}. \forall (\tau_i \in \delta_i(q))_{i \notin I}. \bigcap_{i \in \Sigma} \tau_i \subseteq \sigma$$

$$q \in \text{UPre}_I(\sigma) \text{ iff } \forall (\tau_i \in \delta_i(q))_{i \in I}. \exists (\tau_i \in \delta_i(q))_{i \notin I}. \bigcap_{i \in \Sigma} \tau_i \subseteq \sigma$$

$q \in \text{CPre}_I(\sigma)$ means that, when the system is in state $q$, the team $I$ can force the successor state of $q$ to be in $\sigma$. If $q \in \text{UPre}_I(\sigma)$, it means in state $q$, the team $I$ cannot force the game outside $\sigma$. Of course, if there is only one player, these two operators are equivalent to $\text{pre}$ and $\text{pre}^\ast$.

5.2 Alternating-Time $\mu$-Calculus

$A\mu$ formulas are generated by the grammar:

$$\varphi ::= p \mid \neg p \mid x \mid \varphi_1 \land \varphi_2 \mid \varphi_1 \lor \varphi_2 \mid [\langle I \rangle] \circ \varphi_1 \mid [I] \circ \varphi_1 \mid (\mu x. \varphi_1) \mid (\nu x. \varphi_2)$$
Propositions \( p \) are in a set \( \Pi' \), variables \( x \) are in a set \( X \), and teams \( I \) are in \( \varphi(\Sigma') \).

Given an alternating transition system \( \langle \Sigma, Q, \Delta, \Pi, \pi \rangle \) such that \( \Pi' \subseteq \Pi \) and \( \Sigma = \Sigma' \), with \( \mathcal{E} : X \rightarrow 2^{Q} \) a mapping from the variables to the set of states, each formula \( \varphi \) defines a set of states \( \llbracket \varphi \rrbracket_\mathcal{E} \), computable as follows:

\[
\begin{align*}
\llbracket p \rrbracket_\mathcal{E} &= \pi(p) \\
\llbracket \neg p \rrbracket_\mathcal{E} &= Q \setminus \pi(p) \\
\llbracket x \rrbracket_\mathcal{E} &= \mathcal{E}(x) \\
\llbracket \varphi_1 \land \varphi_2 \rrbracket_\mathcal{E} &= \llbracket \varphi_1 \rrbracket_\mathcal{E} \land \llbracket \varphi_2 \rrbracket_\mathcal{E} \\
\llbracket \varphi_1 \lor \varphi_2 \rrbracket_\mathcal{E} &= \llbracket \varphi_1 \rrbracket_\mathcal{E} \lor \llbracket \varphi_2 \rrbracket_\mathcal{E} \\
\llbracket \{ I \} \circ \varphi_1 \rrbracket_\mathcal{E} &= CPre_1(\llbracket \varphi_1 \rrbracket_\mathcal{E}) \\
\llbracket \{ I \} \circ \varphi_1 \rrbracket_\mathcal{E} &= UPre_1(\llbracket \varphi_1 \rrbracket_\mathcal{E}) \\
\llbracket \mu x. \varphi_1 \rrbracket_\mathcal{E} &= \text{lfp} \lambda \rho. \llbracket \varphi_1 \rrbracket_\mathcal{E}[x \mapsto \rho] \\
\llbracket \nu x. \varphi_1 \rrbracket_\mathcal{E} &= \text{gfp} \lambda \rho. \llbracket \varphi_1 \rrbracket_\mathcal{E}[x \mapsto \rho]
\end{align*}
\]

If \( \varphi \) is closed, \( \llbracket \varphi \rrbracket_\mathcal{E} \) does not depend on \( \mathcal{E} \), and we will write \( \llbracket \varphi \rrbracket \) for \( \llbracket \varphi \rrbracket_\mathcal{E} \).

Given a set of initial states \( I \) and a closed formula \( \varphi \), we will try to approximate \( I \cap \llbracket \varphi \rrbracket \), or \( \text{lfp} \lambda X.(\llbracket \varphi \rrbracket \cap (I \cup \text{Post}(X))) \), rather than \( \text{Post}^*(I) \cap \llbracket \varphi \rrbracket \).

### 5.3 Abstraction of Alternating Transitions Systems

The application of abstract interpretation to alternating transitions systems is already developed in \cite{13}, in a model-checking point of view. The definitions are adapted to our notations as follows: the concrete lattice \( P^\flat \) is here \( \wp(Q) \), so \( \top^\flat = Q \), \( \subseteq^\flat = \subseteq \), \( \bot^\flat = \emptyset \), \( \bigwedge^\flat = \cap \), \( \bigvee^\flat = \cup \). \( P^\sharp \) is the abstract lattice, and there is a Galois connection \( \wp(Q) \xrightarrow{\alpha} P^\sharp \). We define now the abstract operators \( \pi^\sharp \), \( UPre^\sharp_I \) and \( CPre^\sharp_I \).

For each \( p \) in \( \Pi' \), let \( \pi^\sharp(p) \) be an element of \( P^\sharp \) such that \( \pi^\sharp(p) \subseteq \gamma(\pi^\sharp(p)) \), and \( \pi^\sharp(p) \) an element of \( P^\sharp \) such that \( Q \setminus \pi(p) \subseteq \gamma(\pi^\sharp(\overline{p})) \).

For each subset \( I \) of \( \Sigma \), we define the abstract controllable predecessor relation \( CPre^\sharp_I \in P^\sharp \rightarrow P^\sharp \) and the abstract uncontrollable predecessor relation \( UPre^\sharp_I \in P^\sharp \rightarrow P^\sharp \). These relations must satisfy, \( \forall \sigma \subseteq Q: \)

\[
\alpha \circ CPre_1(\sigma) \subseteq^\sharp CPre^\sharp_I \circ \alpha(\sigma) \\
\alpha \circ UPre_1(\sigma) \subseteq^\sharp UPre^\sharp_I \circ \alpha(\sigma)
\]

These operators are not those used in \cite{13} for the abstract model checking algorithm. The authors use under approximation of concrete relation to obtain a sound abstract model checker. In this paper, we use upper approximation.

\footnote{If \( \alpha(\pi(p)) \) is computable, we can take \( \pi^\sharp(p) = \alpha(\pi(p)) \)}
5.4 Combining Forward and Backward Abstractions

We need an abstract successor operator for forward analysis. This abstract successor relation $\text{Post}^\sharp$ must satisfy:

$$\alpha \circ \text{Post}(\sigma) \subseteq^\sharp \text{Post}^\sharp \circ \alpha(\sigma)$$

Again, we define $\text{Post}^{\star\sharp} = \lambda X. \text{lfp} \lambda Y. (X \cup^\sharp \text{Post}^\sharp(Y))$. One can easily check that:

$$\alpha \circ \text{Post}^{\star\sharp}(\sigma) \subseteq^\sharp \text{Post}^{\star\sharp} \circ \alpha(\sigma)$$

We consider the closed $A\mu$ formulas $\varphi$ generated by the grammar:

$$\varphi ::= p \mid \neg p \mid \varphi_1 \lor \varphi_2 \mid \varphi_1 \land \varphi_2 \mid \llbracket I \rrbracket \land \varphi_1 \mid \llbracket I \rrbracket \lor \varphi_1$$

with $\sigma \in \{\mu, \nu\}$. As before, the last term of the grammar can be rewritten exchanging $\lor$ and $\land$ without modifying the expressivity of the logic.

As for the non-game case, we can now define, if $\varphi$ is a formula, $\Omega_\varphi \in P^\sharp \rightarrow P^\sharp$ as follows:

$$\Omega_p(S) = \pi^\sharp(p) \cap^\sharp S$$
$$\Omega_{\neg p}(S) = \pi^\sharp(\overline{p}) \cap^\sharp S$$
$$\Omega_{\varphi_1 \land \varphi_2}(S) = \Omega_{\varphi_1}(S) \cap^\sharp \Omega_{\varphi_2}(S)$$
$$\Omega_{\varphi_1 \lor \varphi_2}(S) = \Omega_{\varphi_1}(S) \cup^\sharp \Omega_{\varphi_2}(S)$$
$$\Omega_{\llbracket I \rrbracket \land \varphi}(S) = \text{CPre}^\sharp_I(\Omega_{\varphi}(\text{Post}^\sharp(S)))$$
$$\Omega_{\llbracket I \rrbracket \lor \varphi}(S) = \text{UPre}^\sharp_I(\Omega_{\varphi}(\text{Post}^\sharp(S)))$$

$$\Omega_{\sigma x. (\varphi \lor \bigvee_{I \in \varphi} \llbracket I \rrbracket \lor \bigvee_{I' \in \varphi} \llbracket I' \rrbracket \land \llbracket I' \rrbracket \lor x)}(S) =$$

$$\operatorname{Llfp} \lambda X. (\lambda X. (S \cup^\sharp \text{Post}^\sharp(X)),$$
$$\lambda Y. (\Omega_{\varphi}(\text{Post}^{\star\sharp}(S))),$$
$$\cup^\sharp \bigcup I (\Omega_{\varphi_1}(\text{Post}^{\star\sharp}(S)) \cap^\sharp \text{CPre}^\sharp(Y))$$
$$\cup^\sharp \bigcup I' (\Omega_{\varphi_1}(\text{Post}^{\star\sharp}(S)) \cap^\sharp \text{UPre}^\sharp(Y))))$$

**Theorem 2.** For all formula $\varphi$ generated by the grammar above, and $I \subseteq Q$:

$$\alpha(I \cap \llbracket \varphi \rrbracket) \subseteq^\sharp \alpha(I) \cap^\sharp \Omega_\varphi(\alpha(I))$$

**Proof.** The proof is essentially the same of the non-game case.

All we need are the equalities $I \cap \text{UCPre}_I(\llbracket \varphi_1 \rrbracket) \subseteq \text{UCPre}_I(\text{Post}(I \cap \llbracket \varphi_1 \rrbracket))$ with $\text{UCPre} = \text{UPre}$ or $\text{CPre}$, and the equivalence:

$$Y \subseteq b^\sharp(X) \iff Y \subseteq b^\sharp(X \cap \text{Post}(Y))$$

with $b^\sharp = \lambda X. (A \cup \bigcup I (B_I \cap \text{CPre}_I(X)) \cup \bigcup I' (C_{I'} \cap \text{UPre}_{I'}(X)))$. These properties are quite easy to check.
6 A Simple Example

We illustrate the combination with a very short and easy example. We will analyze this small non-deterministic program:

\[
\begin{align*}
(0) \{ & x = 1 \} \\
(1) & \text{while (n>0) do} \{ \\
(2) & \text{if (random in [0,1]=0) then} \\
(3) & x = x \times n; \\
(4) & \text{else} \\
(5) & x = x \times (n-1); \\
(6) & \text{fi} \\
(7) & n = n - (\text{input in [0,1]}); \\
(8) & \} \\
(9) &
\end{align*}
\]

Here, \( x, n \) are integers, \((\text{random in [0,1]})\) returns a random integer in \([0,1]\), and \((\text{input in [0,1]})\) returns an integer in \([0,1]\) entered by the user (these commands behave in the same way in the transition relation). Control point (0) is the program entry, we differentiate it from control point (1), which is the while loop entry.

With initial condition \( x=1 \) at control point (0), we will try to prove that the user cannot be sure to have \( x=0 \) at control point (9), that is, the initial condition satisfies \( \nu x. ((\neg A) \land (B \lor \Diamond x) \land (C \lor \Box x)) \) with \( A \) meaning that \( x=0 \) at control point (9), \( C \) being the set of states at control point (2), and \( B \) being the set of states at other control points.

As we use an upper approximation, we take the negation of the proposition, that is (knowing that \( \neg B = C \) : \( \mu x. (A \lor (B \land \Diamond x) \lor (C \land \Box x)) \)). So we must approximate \( \text{lfp} \lambda x. ([A] \cup ([B] \cap \text{pre}(x)) \cup ([C] \cap \text{pre}(x))) \).

We will use interval analysis [6], with the improvement of the results of local decreasing iterations [11] for assignments in the backward analysis.

As we use an upper approximation, we must abstract \( \text{post}(X) \), \( \text{pre}(X) \) and \( \hat{\text{pre}}(X) \). Abstract operators may be described as systems of semantics equations [4, 5]. The program is almost deterministic, and \( \hat{\text{pre}}^z \) is very close to \( \text{pre}^z \). The differences appear at control points (2) and (7), but we only need to express it at control point (2), with the equation:

\[ P_2 = P_3 \cap P_5 \]

(\( \cap \) being the intersection of abstract environments).
The following table gives the results with a single forward analysis ($F^\sharp(\top^\sharp)$), a single backward analysis ($B^\sharp(\top^\sharp)$), the intersection of both analyses ($F^\sharp(\top^\sharp) \cap B^\sharp(\top^\sharp)$), and the first iteration of combination ($B^\sharp(F^\sharp(\top^\sharp))$):

<table>
<thead>
<tr>
<th>Lab. (var.)</th>
<th>$F^\sharp(\top^\sharp)$</th>
<th>$B^\sharp(\top^\sharp)$</th>
<th>$F^\sharp(\top^\sharp) \cap B^\sharp(\top^\sharp)$</th>
<th>$B^\sharp(F^\sharp(\top^\sharp))$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0) x:</td>
<td>[1]</td>
<td>$-\infty, +\infty$</td>
<td>[1]</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>n:</td>
<td>$[-\infty, +\infty]$</td>
<td>$-\infty, +\infty$</td>
<td>$[-\infty, +\infty]$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>(1) x:</td>
<td>[0, +\infty]</td>
<td>$-\infty, +\infty$</td>
<td>[0, +\infty]</td>
<td>$[0]$</td>
</tr>
<tr>
<td>n:</td>
<td>$[-\infty, +\infty]$</td>
<td>$-\infty, +\infty$</td>
<td>$[-\infty, +\infty]$</td>
<td>$[-\infty, +\infty]$</td>
</tr>
<tr>
<td>(2) x:</td>
<td>[0, +\infty]</td>
<td>$-\infty, +\infty$</td>
<td>[0, +\infty]</td>
<td>[0]</td>
</tr>
<tr>
<td>n:</td>
<td>[1, +\infty]</td>
<td>$-\infty, +\infty$</td>
<td>[1, +\infty]</td>
<td>[1, +\infty]</td>
</tr>
<tr>
<td>(3) x:</td>
<td>[0, +\infty]</td>
<td>$-\infty, +\infty$</td>
<td>[0, +\infty]</td>
<td>[0]</td>
</tr>
<tr>
<td>n:</td>
<td>[1, +\infty]</td>
<td>$-\infty, +\infty$</td>
<td>[1, +\infty]</td>
<td>[1, +\infty]</td>
</tr>
<tr>
<td>(4) x:</td>
<td>[0, +\infty]</td>
<td>$-\infty, +\infty$</td>
<td>[0, +\infty]</td>
<td>[0]</td>
</tr>
<tr>
<td>n:</td>
<td>[1, +\infty]</td>
<td>$-\infty, +\infty$</td>
<td>[1, +\infty]</td>
<td>[1, +\infty]</td>
</tr>
<tr>
<td>(5) x:</td>
<td>[0, +\infty]</td>
<td>$-\infty, +\infty$</td>
<td>[0, +\infty]</td>
<td>[0]</td>
</tr>
<tr>
<td>n:</td>
<td>[1, +\infty]</td>
<td>$-\infty, +\infty$</td>
<td>[1, +\infty]</td>
<td>[1, +\infty]</td>
</tr>
<tr>
<td>(6) x:</td>
<td>[0, +\infty]</td>
<td>$-\infty, +\infty$</td>
<td>[0, +\infty]</td>
<td>[0]</td>
</tr>
<tr>
<td>n:</td>
<td>[1, +\infty]</td>
<td>$-\infty, +\infty$</td>
<td>[1, +\infty]</td>
<td>[1, +\infty]</td>
</tr>
<tr>
<td>(7) x:</td>
<td>[0, +\infty]</td>
<td>$-\infty, +\infty$</td>
<td>[0, +\infty]</td>
<td>[0]</td>
</tr>
<tr>
<td>n:</td>
<td>[1, +\infty]</td>
<td>$-\infty, +\infty$</td>
<td>[1, +\infty]</td>
<td>[1, +\infty]</td>
</tr>
<tr>
<td>(8) x:</td>
<td>[0, +\infty]</td>
<td>$-\infty, +\infty$</td>
<td>[0, +\infty]</td>
<td>[0]</td>
</tr>
<tr>
<td>n:</td>
<td>[0, +\infty]</td>
<td>$-\infty, +\infty$</td>
<td>[0, +\infty]</td>
<td>[0]</td>
</tr>
<tr>
<td>(9) x:</td>
<td>[0, +\infty]</td>
<td>[0]</td>
<td>[0]</td>
<td>[0]</td>
</tr>
<tr>
<td>n:</td>
<td>$[-\infty, +\infty]$</td>
<td>$-\infty, +\infty$</td>
<td>$[-\infty, +\infty]$</td>
<td>$[-\infty, +\infty]$</td>
</tr>
</tbody>
</table>

The next iteration of the combination will lead to $\emptyset$ everywhere, which is of course the abstract fixpoint $L^\sharp$. So $L^\flat = \emptyset$ (which is not equal to $F^\flat(\top^\flat) \cap B^\flat(\top^\flat)$). As, for this kind of temporal property, $L^\flat \cap I = I \cap B^\flat(\top^\flat)$, we obtained the expected result.

7 Conclusion

We have proved that the combination of forward and backward analyses still holds to check complex temporal properties. Whereas this combination is useless when dealing with finite domains, and not very useful when abstractions are done by hand (as in the model-checking approach), we expect it will significantly enhance results given by an automatic abstract analyzer of temporal properties.

Using the results of this article will require to have over-approximations of the $\preceq$ operator (or predecessor operators of game logic), something which has not been not much studied until now. We also need a method to compute over-approximations of greatest fixpoints since lower narrowing operators give poor results.
Acknowledgements

I wish to thank the anonymous referees, as well as Radhia and Patrick Cousot, Francesco Logozzo for their comments and suggestions.

References

Abstract. In this paper we design abstract domains for numerical power analysis. These domains are conceived to discover properties of the following type: “The integer (or rational) variable $X$ at a given program point is the numerical power of $c$ with the exponent having a given property $\pi$”, where $c$ and $\pi$ are automatically determined. A family of domains is presented, two of these suppose that the exponent can be any natural or integer value, the others include also the analysis of properties of the exponent set. Relevant lattice-theoretic properties of these domains are proved such as the absence of infinite ascending chains and the structure of their meet-irreducible elements. These domains are applied in the analysis of integer powers of imperative programs and in the analysis of probabilistic concurrent programming, with probabilistic non-deterministic choice.

Keywords: Abstract interpretation, static program analysis, numerical power analysis, probabilistic analysis.

1 Introduction

Abstract interpretation is a general theory for approximating the semantics of programming languages, including static program analysis as a special case. The design of a static program analyzer consists in the design of an approximate decidable semantics, called the abstract semantics, which is systematically derived from the concrete (standard) semantics of the language. This approach has several well known advantages with respect to other methods: (1) The analysis is fully described and constructively derived by the way the concrete data and control flows are approximated; (2) Its correctness with respect to the concrete semantics can be immediately proved formally by construction; (3) New and more advanced analyses can be systematically conceived by modifying the abstraction methods. The analysis consists here in the solution of a system of fixpoint equations associated with the concrete semantics of the program, where each equation is interpreted in an abstract domain returning an approximated transformation of the program invariant at every program point. In order to make the analysis effective, the iterative solution of the approximated system of equations has to terminate. This may be achieved either statically, by
designing suitable abstract domains with no ascending chains, or dynamically while fixpoints are computed by using widening/narrowing operations to speed-up convergence \( \mathcal{R} \). In any case, the design of the abstract domain is central in the construction of any abstract interpretation, and therefore of any static program analysis algorithm. An abstract domain is a set of mathematical objects representing concrete properties of programs, e.g. the properties of its data-structures. In the standard adjoint framework of abstract interpretation \( \mathcal{I} \), a pair of adjoint functions relates the concrete and abstract domains in such a way that the unique best (i.e. most precise) approximated property can always be associated with each concrete object by means of an abstraction function. This is an ideal situation which provides the designer of the analysis of a number of powerful mathematical tools for proving the correctness and in some cases the optimality of the analysis.

The main results. The aim of this paper is to present a family of domains for numerical power analysis, namely the power analysis of numerical (integer or rational) values in the framework of abstract interpretation. These domains are conceived to discover properties of the following type: “The integer (or rational) variable \( X \) at a given program point is the numerical power of \( c \) with the exponent having a given property \( \pi \)”, where \( c \) and \( \pi \) are automatically determined: \( c \) is an integer or rational number and \( \pi \) is a property of natural or integer numbers. Consider for instance the following program \( P \), also known as the Collatz problem [14]:

```plaintext
while \( n \neq 1 \) do
    \( \pi_2 : \)
    \( n := \text{if even}(n) \text{ then } n/2 \text{ else } 3n + 1 \)
endw
```

It is immediate that \( \{ \exists k. n = c^k \} P \{ n = 1 \} \) is a valid Hoare triple, with \( c \) being any power of 2, e.g. \( c = 2 \); namely the program \( P \) terminates with \( n = 1 \) whenever the input \( n = 2^k \) holds. The numerical power analysis is devoted to automatically discover these situations.

We present a family of domains for numerical power analysis, denoted \( \mathbb{P}(\mathbb{B}, \mathbb{E}) \), which are parametric in the set of possible exponents. In particular \( \mathbb{B} \in \{ \mathbb{Z}, \mathbb{Q} \} \) while \( \mathbb{E} \) can be a generic abstract domain on \( \mathbb{N} \) or \( \mathbb{Z} \), with specific properties that we will define in the following. Clearly if \( \mathbb{E} \) is an abstract domain of integers then \( \mathbb{B} = \mathbb{Q} \). We distinguish two cases: (1) when \( \mathbb{E} = \{ X \} \) and \( X \in \{ \mathbb{N}, \mathbb{Z} \} \), the domain \( \mathbb{P}(\mathbb{B}, \mathbb{E}) \) determines whether a variable has a value of the kind \( c^k \) with \( c \in \mathbb{B} \) and \( k \) is any value in \( X \); (2) when \( \mathbb{E} \) it self is an abstract domain, then we are able to analyze the properties of the exponent set too. All the domains of this family are fully specified and proved correct in the standard Cousot’s adjoint framework of abstract interpretation. Moreover, they share relevant lattice-theoretic properties. In particular, while all these domains may have infinite descending chains, they all satisfy the ascending chain condition ACC (no infinite ascending chains are permitted). This holds when the domain \( \mathbb{E} \) is it self an ACC domain.
This is particularly important in static program analysis as it proves that the corresponding analyses are all decidable (no infinite loops are allowed in the abstract semantics) and no widening is necessary to achieve termination. The structure of these domains is presented by providing a characterization of their lattice-theoretic structure in terms of its meet-irreducible elements. The abstract interpretation of standard arithmetic operations on numerical powers are proved correct and optimal.

The fact that a program variable belongs to a numerical power of the form $c^k$ is a quite rare event in standard programming. This is justified formally by the fact that few operations allows a variable to maintain its values into a given numerical power. However, as it is often the case for domains approximating numerical values, numerical power analysis becomes particularly informative when combined with other domains for the analysis of numerical values, like interval analysis [1]. This is the case in all the applications where the detection of program invariants that involve numerical powers at a given control point contributes to program optimization, e.g. to detect the upper/lower bound in memory allocation in hardware design specification by high-level programming languages (e.g. VHDL specification). An important field of application of numerical power analysis is the static analysis of probabilistic programming languages, where non-deterministic choice is replaced by a randomized choice. In this case, if we have a trace of intermediate states $s_i$ with probability $p_i \in [0, 1]$: $\langle s_0, p_0 \rangle \rightarrow \ldots \rightarrow \langle s_i, p_i \rangle \rightarrow \langle s_{i+1}, p_{i+1} \rangle \rightarrow \ldots$, then the probability of the terminal state (if any) is the product $\prod_i p_i$. It is often the case that the resulting probability is a numerical power of some fraction. The numerical power analysis is accurate for approximating rational product, therefore it can be used to approximate the probability resulting in a randomized computation. We apply this idea to approximate the probability of probabilistic concurrent constraint programs.

**Related works.** A number of domains for static analysis of numerical values have been proposed in the literature. These include constant propagation [21], Granger’s arithmetical congruence analysis [7], interval analysis [1], affine relations [13], linear restraints [4], linear congruence relation analysis [9], and trapezoid congruence analysis [15]. None of these domains include numerical power analysis. Like congruence analysis, numerical power analysis belongs to the family of non-convex domains, namely of those domains whose objects describe a non-convex set of numbers. The relation with numerical congruence analysis is even stronger, as congruence analysis can be easily generalized to the family of abstract domains generated by suitable subgroups of abelian groups. This is not the case of numerical power, as numerical power analysis cannot be reconstructed as a special case of Granger’s congruence analysis, namely as residue class (or congruence class) of a suitable commutative group. This is due for $\mathbb{P}(\mathbb{Z}, \{\mathbb{N}\})$ to the fact that integers with multiplication does not form a group. Similar observations hold for $\mathbb{P}(\mathbb{Q}, \{\mathbb{N}\})$. In his PhD thesis [8], Granger’s introduced the abstract domain of multiplicative congruences over $\mathbb{Q}$. This analysis is essentially an analysis of rational powers with integer exponents, namely the
domain $\mathbb{P}(\mathbb{Q}, \{\mathbb{Z}\})$. We include this domain in our work as special cases of a more general pattern $\mathbb{P}(\mathbb{Q}, \mathbb{E})$ with $\mathbb{E}$ being an abstract interpretation of $\varphi(\mathbb{Z})$.

2 Preliminaries

2.1 Basic Mathematical Notions

If $S$ and $T$ are sets, then $\varphi(S)$ denotes the power-set of $S$, $S\setminus T$ denotes the set-difference between $S$ and $T$, $S \subseteq T$ denotes strict inclusion, and for a function $f : S \rightarrow T$ and $X \subseteq S$, $f(X) \overset{\text{def}}{=} \{f(x) \mid x \in X\}$. By $g \circ f$ we denote the composition of the functions $f$ and $g$, i.e., $g \circ f \overset{\text{def}}{=} \lambda x. g(f(x))$. The notation $\langle P, \leq \rangle$ denotes a poset $P$ with ordering relation $\leq$, while $\langle P, \leq, \vee, \wedge, \top, \bot \rangle$ denotes a complete lattice $P$, with ordering $\leq$, least upper bound ($\text{lub}$ for short) $\vee$, greatest lower bound ($\text{glb}$ for short) $\wedge$, greatest element (top) $\top$, and least element (bottom) $\bot$. If $P$ is a poset with bottom $\bot$ then $a \in P$ is an atom if $a$ covers $\bot$, i.e., $\bot < a$ and for all $x \in P$, $\bot < x \leq a$ implies $x = a$. Dual-atomic elements are dually defined. An element $x \in P$ is meet-irreducible if $x = a \wedge b \Rightarrow x \in \{a, b\}$. The set of meet-irreducible elements in $P$ is denoted $\text{Mirr}(P)$. A poset is ACC if for each ascending chain $\{x_1 \leq x_2 \leq \ldots \leq x_n \leq \ldots\}$ there is $k \in \mathbb{N}$ such that $\forall n \geq 0: x_k = x_{k+n}$, i.e. every ascending chain has finite limit. A DCC poset is dually defined. Consider $S \subseteq P$ then the downward closure of $S$ is defined as $\downarrow S \overset{\text{def}}{=} \{x \in P \mid \exists y \in S : x \leq_P y\}$. If $X$ is a set then $(X)^n$ denotes the set of its upper bounds and $|X|$ the number of its elements. If $n \in \mathbb{Z}$ then $|n|$ represent its modulus. If $k, n \in \mathbb{N}$ and $\exists k' \in \mathbb{N}, n = k \cdot k'$ then we write $k|n$. If $m \in \mathbb{N}$ is the least common multiple of two values $k, h \in \mathbb{N}$ then we write $m = \text{lcm}(k, h)$ and if it is the greatest common divisor of them we write $m = \gcd(k, h)$.

2.2 Abstract Interpretation

Abstract domains can be equivalently formulated either in terms of Galois connections or closure operators [2]. An upper closure operator on a poset $P$ is an operator $\rho : P \rightarrow P$ which is monotone, idempotent, and extensive ($\forall x \in P, x \leq_P \rho(x)$). The set of these operators is denoted by $\text{uco}(P)$. Let $\langle C, \leq, \vee, \wedge, \top, \bot \rangle$ be a complete lattice. A basic property of closure operators is that each closure is uniquely determined by the set of its fix-points $\rho(C)$. When $C$ is a complete lattice then both $\langle \text{uco}(C), \subseteq, \cup, \cap, \lambda x. \top, \lambda x. x\rangle$ and $\langle \rho(C), \leq, \vee, \wedge, \top, \rho(\bot)\rangle$ are complete lattices where $\forall \rho, X = \rho(\lambda x. x)$. $X \subseteq C$ is the set of fix-points of an upper closure on $C$ iff $X$ is a Moore-family of $C$, i.e., $X = \mathcal{M}(X) \overset{\text{def}}{=} \{\wedge S : S \subseteq X\}$ — where $\wedge \emptyset = \top \in \mathcal{M}(X)$. For any $X \subseteq C$, $\mathcal{M}(X)$ is called the Moore-closure of $X$ in $C$, i.e., $\mathcal{M}(X)$ is the least (w.r.t. set-inclusion) subset of $C$ which contains $X$ and is a Moore-family of $C$. We say that $C$ is meet-generated by its meet-irreducible elements iff $C = \mathcal{M}(\text{Mirr}(C))$. If $A$ and $C$ are posets, and $\alpha : C \overset{\text{m}}{\rightarrow} A$ and $\gamma : A \overset{\text{m}}{\rightarrow} C$ are monotone functions such that $x \leq \gamma(\alpha(x))$ and $\alpha(\gamma(x)) \leq x$, then the quadruple $(A, \alpha, \gamma, C)$ is called a Galois connection (GC for short) or adjunction between $C$ and $A$. Note that in GC for any
\( x \in C \) and \( y \in A \): \( \alpha(x) \leq_A y \Leftrightarrow x \leq_C \gamma(y) \) and \( \gamma(y) = \bigvee \{ x \mid \alpha(x) \leq y \} \) and \( \alpha(x) = \bigwedge \{ y \mid x \leq \gamma(y) \} \). If in addition \( \alpha \circ \gamma = \lambda x.x \), then \((A, \alpha, \gamma, C)\) is a Galois insertion (GI) of \( A \) in \( C \). In this setting, the concrete and abstract domains, respectively \( C \) and \( A \), are assumed to be complete lattices and are related by abstraction and concretization maps forming a GC \((A, \alpha, \gamma, C)\) \cite{1}. Following a standard terminology, \( A \) is called abstraction of \( C \), and \( C \) is a concretization of \( A \). If \((A, \alpha, \gamma, C)\) is a GI, then each value of the abstract domain \( A \) is useful in representing \( C \), because all the elements of \( A \) represent distinct members of \( C \), being \( \gamma \) 1-1. Any GC may be lifted to a GI by identifying in an equivalence class those values of the abstract domain with the same concretization. This process is known as reduction of the abstract domain. The following result relates GI’s with closure operators, and so with Moore families.

**Theorem 1** (\cite{2}). Let \( C \) and \( A \) two complete lattices, then \((A, \alpha, \gamma, C)\) is a GI iff \( A \) is isomorphic to a Moore family of \( C \).

Let \( \{A_i\}_{i \in I} \subseteq uco(C) \): \( \cap_{i \in I} A_i \) is (isomorphic to) the reduced product (basically cartesian product plus reduction) of all the \( A_i \)’s, or, equivalently, it is the most abstract domain which is more concrete than every \( A_i \). Let us remark that the reduced product can be also characterized as Moore-closure of set-union, i.e. \( \cap_{i \in I} A_i = M(\cup_{i \in I} A_i) \). Let \( Program \) denote the set of (syntactically well-formed) programs. The concrete standard semantics is a function \([\cdot] : Program \rightarrow C\), where \( C \) is a concrete semantic domain of denotations, which we assume to be a complete lattice. If an abstract interpretation is specified by a GI \((A, \alpha, \gamma, C)\) and by an abstract semantic function \([\cdot]^\sharp : Program \rightarrow A\), then \([\cdot]^\sharp\) is a sound abstract semantics, or (correctly) approximates \([\cdot]\) if, for any program \( P \), \( \alpha([P]) \leq_A [P]^\sharp \), or, equivalently, \([P] \leq_C \gamma([P]^\sharp)\). When concrete semantics are specified in fix-point form, i.e. \( [P] = \text{lfp}(F_P) \in C \) for some given concrete domain \( C \) and semantic operator \( F_P \), then given a corresponding abstract semantics \( S^\sharp = \langle A, F_P^\sharp \rangle \) and a GI \((A, \alpha, \gamma, C)\), \( S^\sharp \) is called a sound abstraction of \( S \) if for all \( P \in Program\), \( \alpha(\text{lfp}(F_P)) \leq_A \text{lfp}(F_P^\sharp) \). This soundness condition can be more easily verified point-wise by checking whether for all \( P \in Program\), \( \alpha \circ F_P \leq_A F_P^\sharp \circ \alpha \), or, equivalently, \( \alpha \circ F_P \circ \gamma \leq_C F_P^\sharp \). If \( \alpha \circ F_P \circ \gamma = F_P^\sharp \) then \( F_P^\sharp \) is optimal \cite{2}.

### 3 The Domain of Numerical Powers

In this section we introduce a class of domains for power analysis of numerical variables. Consider the following program fragment:

\[
\begin{align*}
n &:= 1; \\
\text{while } n \leq 100 &\text{ do} \\
&n := n * 2 \\
\text{endw}
\end{align*}
\]

This program computes the smallest power of 2 which is greater than 100, namely \( n = 128 \). In order to automatically discover the invariant of this program, i.e.
\( Inv = \{ n = 2^k \land k \in \mathbb{N} \land n \in [1, 128] \} \) we need to design a domain whose objects represent numerical powers of integer numbers. Variables have both values in \( \mathbb{Z} \) and \( \mathbb{Q} \), therefore the collection of possible values that a variable may assume during the execution of a program is an element in \( \wp(\mathbb{Z}) \) and \( \wp(\mathbb{Q}) \). Recall that \( \langle \wp(X), \subseteq, \mathbb{Z}, \emptyset, \cap, \cup \rangle \) with \( X \in \{ \mathbb{Z}, \mathbb{Q} \} \) is a complete Boolean lattice, representing the concrete domain of interpretation. We consider standard numerical types: natural, integer and rational. We identify the elements that we consider useful for the kind of analysis that we are going to make, namely we define the collection of sets each one containing natural or integer powers of a particular integer or rational value. Let \( \mathbb{B} \in \{ \mathbb{Z}, \mathbb{Q} \} \) and \( E \in \{ \wp_X \subseteq \wp(X) | X \in \{ \mathbb{N}, \mathbb{Z} \} \} \). Consider \( a \in \mathbb{B} \) and \( X \in E \), then we define the set of the \( X \)-powers of \( a \) as \( \{ a \}^X \overset{\text{def}}{=} \{ a^k | k \in X \} \). In the following if \( X \) is a numerical set we will use the notation \( X_0 \) for \( X \setminus \{ 0 \} \).

The following proposition is immediate.

**Proposition 1.** Let \( a, b \in \mathbb{B} \) and \( k \in \mathbb{Z}_0 \) such that \( a = b^k \) then \( \{ a \}^X = \{ b \}^{kX} \).

**Definition 1.** Let \( X \in \{ \mathbb{N}, \mathbb{Z}, \mathbb{Q} \} \) and \( a \in X \). The exponential atom (e-atom) of \( a \) is the least value \( b \in X \) such that \( a = b^k \) for some \( k \in \mathbb{N}_0 \). The (unique) e-atom of \( a \) is denoted \( a_1 \). The value \( a \) is atomic if \( a = a_1 \).

**Corollary 1.**
- Let \( X \in E = E_\mathbb{Z} \) and \( a \in \mathbb{Q} \) then \( \{ a \}^X = \{ a^{-1} \}^{-X} \).
- Let \( X \in E \) and \( a \in \mathbb{B} \) then \( \{ a \}^X = \{ a_1 \}^{kX} \) where \( k \in \mathbb{N} \) is such that \( a = a_1^k \).

At this point we can introduce the class of domains which include all the \( X \)-powers of integer or rational values.

**Definition 2.** \( \wp(\mathbb{B}, E) \overset{\text{def}}{=} \{ \emptyset, \mathbb{B} \} \cup \{ \{ a \}^X | a \in \mathbb{B} \land X \in E \} \), with the condition that if \( E = E_\mathbb{Z} \) then \( \mathbb{B} = \mathbb{Q} \).

Corollary 1 tells us that in \( \wp(\mathbb{B}, E) \) we can first work only on the sets of powers of atomic elements and then, under particular conditions, extend all the results obtained to the whole \( \wp(\mathbb{B}, E) \). In particular, when \( \mathbb{B} = \mathbb{Q} \) and \( E = E_\mathbb{Z} \), we can work only with sets of powers of values greater than one and then we can naturally extend all the results to the sets of power of all the rational values. Hence, in the following, when we deal with atomic elements in \( \mathbb{Q} \), we consider them always as greater than one. Note also that \( \wp(\mathbb{B}, E) = \{ \emptyset, \mathbb{B} \} \cup \{ \{ a \}^X | a \in \mathbb{B}^+ \land X \in E \} \cup \{ \{ a \}^X | a \in \mathbb{B}^- \land X \in E \} \) is the coalesced sum of two disjoint domains with common top and bottom elements, \( \mathbb{B} \) and \( \emptyset \) respectively. Namely we can see the domain \( \wp(\mathbb{B}, E) \) as composed by two disjoint sub-domains, one with positive bases and one with negative bases. We can also note that if \( a \in \mathbb{B}^+ \) and \( X \in E \) then \( \{ |b| | b \in \{ a \}^X \} = \{ |b| | b \in \{ -a \}^X \} \) but \( \{ a \}^X \neq \{ -a \}^X \). This means that the elements \( \{ a \}^X \) and \( \{ -a \}^X \) are constituted by the same values but the sign. This symmetry is important because it allows us to extend any property that we are going to prove on \( \{ \emptyset, \mathbb{B} \} \cup \{ \{ a \}^X | a \in \mathbb{B}^+ \land X \in E \} \) to the whole domain \( \wp(\mathbb{B}, E) \). The following lemma says that, if there exists \( h \in X_0 \) and \( k \in Y_0 \) such that \( a^k = b^h \), and both \( a \) and \( b \) are atomic, then \( a = b \).
Lemma 1.

1. Let $a, b \in B$ be atomic with $a \neq b$ and $X, Y \in \wp(X)$, $X \in \{\mathbb{N}, \mathbb{Z}\}$, then
\[
\{a\}^X \cap \{b\}^Y = \begin{cases} 
\{1\} & \text{if } 0 \in X \cap Y \\
\emptyset & \text{otherwise}
\end{cases}
\]

2. If $a \in B$ is atomic and $(E, \subseteq)$ has a top element $\top$ then $\{a\}^\top$ is a dual-atom of $\mathbb{P}(B, E)$.

This result allows us to say that all the domains $(\downarrow \{a\}^\top, \subseteq)$, when $a$ is atomic, are completely disjoint, i.e. they have no elements in common. Figure 1 shows the domain $\mathbb{P}(\mathbb{Z}, \{\mathbb{N}\})$.

4 Analyzing the Base

In this section we focus our attention on the analysis of the base of numerical powers only. This is achieved by considering that exponents may have any natural or integer value. We introduce two basic abstract domains $\mathbb{P}(B, \{X\})$, where $X \in \{\mathbb{N}, \mathbb{Z}\}$ and we prove their algebraic properties. In particular, we would like to find that they form complete lattices and are both an abstraction of $\wp(B)$. In this way we would be able to analyze the described property for any integer and rational variable. First we have to prove that $\mathbb{P}(B, \{X\})$ is a lattice (we define the finite lub and glb operations) and then, by showing a GI between $\wp(B)$ and $\mathbb{P}(B, \{X\})$, we can conclude, by Theorem 1, that $\mathbb{P}(B, \{X\})$ is a Moore family of $\wp(B)$. Therefore it is a complete lattice and, in particular, an abstraction of $\wp(B)$.

Lemma 2. Let $X \in \{\mathbb{N}, \mathbb{Z}\}$ and $k, h \in X$ then $kX \cap hX = \text{lcm}(k, h)X$.

Theorem 2. Let $a, b \in B$ such that $a = a^k\uparrow$, $b = b^h\uparrow$ for some $k, h \in \mathbb{N}$. Consider $X \in \{\mathbb{N}, \mathbb{Z}\}$
\[
\{a\}^X \cap \{b\}^X = \begin{cases} 
\{d\text{lcm}(k, h)^X\} & \text{if } a^\uparrow = b^\uparrow = d \\
\{1\} & \text{otherwise}
\end{cases}
\]

Proof. Consider $a^\uparrow = b^\uparrow = d$
\[
\{d^k\}^X \cap \{d^h\}^X = \{d\}^{kX} \cap \{d\}^{hX} = \{d^w \mid w \in kX\} \cap \{d^w \mid w \in hX\} = \\
= \{d^w \mid w \in kX \land w \in hX\} = \{d^w \mid w \in kX \cap hX\} = \\
= \{d\}^{kX \cap hX} = \{d\}^{\text{lcm}(k, h)X} = \{d\text{lcm}(k, h)^X\}
\]

for Corollary 1 and Lemma 2. If $a^\uparrow \neq b^\uparrow$, then the intersection is $\{1\}$ for what we proved in Lemma 1 (point 1.) because $\{a\}^X \cap \{b\}^X = \{a\}^{kX} \cap \{b\}^{hX} = \{1\}$.

Let $B = \mathbb{Z}$ and $X = \mathbb{N}$, then, in practice, for finding the intersection of two sets of powers, $\{a\}^\mathbb{N}$ and $\{b\}^\mathbb{N}$, we have to find the e-atoms of the values $a$ and $b$. For example, consider $a$ and suppose that its factorization is $\prod_{i=1}^s p_i^{m_i}$ then
we take \( k = \gcd(m_1, \ldots, m_s) \) which is such that \( a = a_1^k \), and if we assume \( \forall i \leq s. n_i = m_i/k \) then \( a \uparrow = \prod_{i=1}^s p_i^{n_i} \). Intuitively this value \( k \) is the greatest exponent that is common to all the exponents \((m_i)\) in the factorization. In this way the value whose prime factorization is obtained by erasing the value \( k \) from the exponents \((\prod_{i=1}^s p_i^{n_i})\), is the e-atom of \( a \) because it is the least value, by construction, such that \( a \) is a power of it. The same we can do for \( b \) finding that for some \( h \in \mathbb{N} \) we have \( b = b_1^h \). Now if \( a_\uparrow = b_\uparrow \) we find the intersection as \( \{a\}^N \cap \{b\}^N = \{a_\uparrow \lcm(k,h)\}^N \).

**Example 1.** Consider \( a, b \in \mathbb{Z} \), we have to determine \( c \in \mathbb{Z} \) such that \( \{c\}^N = \{a\}^N \cap \{b\}^N \) where \( a = 144 = 2^4 \cdot 3^2 \) and \( b = 1728 = 2^6 \cdot 3^3 \). Then, using the notation of the theorem, we have that \( k = \gcd(2, 4) = 2 \) so \( a_\uparrow = 2^2 \cdot 3 \), namely the same prime factors of \( a \) with the exponents divided for their \( \gcd \). Moreover \( h = \gcd(6, 3) = 3 \) so \( b_\uparrow = 2^2 \cdot 3 \), hence the two values share the same e-atom and therefore the intersection exists. So we can easily see that \( a = (2^2 \cdot 3^2) \), \( b = (2^2 \cdot 3^3) \) and \( \lcm(k, h) = \lcm(2, 3) = 6 \). At this point we can conclude that \( c = (2^2 \cdot 3^3)^6 = 2985984 \) is the value which identifies the intersection.

When \( \mathbb{B} = \mathbb{Q} \), \( \mathbb{X} \in \{\mathbb{N}, \mathbb{Z}\} \), and we want to find the intersection of two sets of powers \( \{p\}^\mathbb{X} \) and \( \{q\}^\mathbb{X} \), we have to calculate the e-atoms of the values \( p \) and \( q \). First of all if \( \mathbb{X} = \mathbb{Z} \) and \( p \leq 1 \) then we consider \( \{p\}^\mathbb{Z} = \{1/p\}^\mathbb{Z} \), the same we can do if \( l \leq 1 \). For this reason we can suppose both \( p \) and \( q \) greater than one. Consider \( p = a/b \), then it is clear that \( p = a_\uparrow^{k_1}/b_\uparrow^{k_2} \) for some \( k_1, k_2 \in \mathbb{N} \). Now if \( m = \gcd(k_1, k_2) \), then we can rewrite the equality as \( p = a_\uparrow^{k_1/m}/b_\uparrow^{k_2/m} \) where \( k_1 = k_1'm \) and \( k_2 = k_2'm \). Hence \( p = p_\uparrow^m \) and \( p_\uparrow = a_\uparrow^{k_1}/b_\uparrow^{k_2} \). The same we can do for \( q \) and then we calculate the intersection as we have done in the example before.

**Example 2.** Let \( p = \frac{54^4}{5^8} = \frac{(2 \cdot 3^3)^4}{5^8} \) and \( q = \frac{54^6}{5^{12}} = \frac{(2 \cdot 3^3)^6}{5^{12}} \). We can see that \( k = \gcd(4, 8) = 4 \) and \( h = \gcd(6, 12) = 6 \), then \( p_\uparrow = \frac{2 \cdot 3^3}{5^2} = q_\uparrow^{\text{def}} = r \), which are the product of the same prime factors of \( p \) with the exponents divided for their \( \gcd \). Now we can find the intersection of the two values, \( p = r^4 \) and \( q = r^6 \), as \( r^m \) where \( m = \lcm(4, 6) = 12 \). Hence the intersection is represented by \( r^{12} = \left(\frac{2 \cdot 3^3}{5^2}\right)^{12} \).

As for the least upper bound operation is concerned, we consider the following theorem.

**Theorem 3.** Let \( a, b \in \mathbb{B} \) such that \( a = a_1^k \), \( b = b_1^h \) for some \( k, h \in \mathbb{N} \). Consider \( \mathbb{X} \in \{\mathbb{N}, \mathbb{Z}\} \)

\[
\{a\}^\mathbb{X} \lor \{b\}^\mathbb{X} = \begin{cases} 
\{d_\gcd(k,h)\}^\mathbb{X} & \text{if } a_\uparrow = b_\uparrow = d \\
\mathbb{B} & \text{otherwise}
\end{cases}
\]

**Proof.** Consider \( a_\uparrow = b_\uparrow = d \). Then surely \( \{a\}^\mathbb{X} \), \( \{b\}^\mathbb{X} \subseteq \{d_\gcd(k,h)\}^\mathbb{X} \) because both \( a \) and \( b \) are equal to a power of \( d_\gcd(k,h) \). Consider now \( \{c\}^\mathbb{X} \supseteq \{a\}^\mathbb{X} \), \( \{b\}^\mathbb{X} \)
with \(c \in \mathbb{B}\). Then we can say that \(\exists k' \in \mathbb{X}, a = d^k = c^{k'}\) and \(\exists h' \in \mathbb{X}, b = d^h = c^{h'}\) for definition of inclusion. Because \(d\) is atomic, this equality implies that \(c_1 = d\) by Lemma 1, so there exists \(z \in \mathbb{X}\) such that \(c = d^z\). But \(z\) must be a divisor of \(k\) and of \(h\) for the equalities written above. Then by definition of \(gcd\), this implies that \(z \leq gcd(k, h)\), namely \(c \leq d^{gcd(k, h)}\). Now we have to prove that \(\{c\}^X\) contains the candidate to be the least upper bound, a way is to prove that \(d^{gcd(k, h)}\}^X \cap \{c\}^X = \{d^{gcd(k, h)}\}^X\). Note that \(a, b \in \{d^{gcd(k, h)}\}^X \cap \{c\}^X = \{e\}^X\) for some \(e \in \mathbb{B}\). This implies that \(\{a\}^X, \{b\}^X \subseteq \{e\}^X\) and we have just seen that this implies that \(e \leq d^{gcd(k, h)}\). For the properties of intersection we have also that \(\{e\}^X \subseteq \{d^{gcd(k, h)}\}^X\), so \(e \geq d^{gcd(k, h)}\). We can conclude that it must be \(e = d^{gcd(k, h)}\), namely \(\{e\}^X \supseteq \{d^{gcd(k, h)}\}^X\).

As in the intersection we can describe an analogous constructive method for systematically derive the representative value of the least upper bound of two generic elements of the domain.

**Example 3.** Consider \(a, b \in \mathbb{Z}\), \(a = 3^{12} \cdot 5^6\) and \(b = 3^{16} \cdot 5^8\), we want to find the value which represents \(\{a\}^N \lor \{b\}^N\). As in Example 1 consider \(k = gcd(6, 12) = 6\) and \(h = gcd(8, 16) = 8\), then we can calculate the common e-atom of the two numbers as \(a_1 = 3^2 \cdot 5 = b_1 = d\), which is, for example, the value \(a\) with the exponents of the factorization divided by \(k = 6\). Then \(a = d^6\) and \(b = d^8\) and, by using the notation of the theorem, we calculate \(m = gcd(6, 8) = 2\) and we can conclude that the least upper bound is \(\{3^4 \cdot 5^2\}^N\).

![Fig. 1. Integer's power domain \(\mathbb{P}(\mathbb{Z}, \{N\})\).](image-url)
5 Abstracting the Exponent Set

In this section we enhance our domain in order to better characterize the exponent set of numerical powers. This is achieved by observing that, in Definition 2, $\mathbb{E}$ may be any collection of sets of integers or natural numbers. In particular, $\mathbb{E}$ can be an abstraction of $\varphi(\mathbb{N})$ or $\varphi(\mathbb{Z})$. The resulting domain is more precise than the one described above because it allows us to analyze numerical powers by combining the domain in Section 4 for analyzing the bases and other abstract domains, like interval analysis, congruence analysis etc., for analyzing the exponents. As we will see later on, this generalization is not always possible, and some restrictions on the structure of $\mathbb{E}$ have to be taken into account. Assume that $\mathbb{E}_N \in uco(\varphi(\mathbb{N}))$ and $\mathbb{E}_Z \in uco(\varphi(\mathbb{Z}))$ be respectively abstract domains for natural or integer values, and $\mathbb{E} \in \{\mathbb{E}_X | X \in \{\mathbb{N}, \mathbb{Z}\}\}$. The main problem with this generalization is that if $X \in \mathbb{E}$ and $k \in \mathbb{N}$ (or $k \in \mathbb{Z}$), then in general $kX$ may not belong to $\mathbb{E}$. In order to obtain a Moore family we need that the intersection of numerical powers $\{a\}^X$ and $\{b\}^Y$ has the form of $\{c\}^Z$ with $Z \in \mathbb{E}$. The following definition introduces the notion of exponential-closed domains. These domains ensure that the abstraction of the exponent in a non-trivial domain (i.e. where $\mathbb{E} \neq \{\mathbb{N}\}, \{\mathbb{Z}\})$ is a Moore family, and therefore an abstraction of $\varphi(\mathbb{B})$.

**Definition 3.** Let $X \in \{\mathbb{N}, \mathbb{Z}\}$. A Moore family $\mathbb{E}_X$ is called exponential-closed domain if $\forall X, Y \in \mathbb{E}_X \land \forall k, h \in X. \exists z \in X, W \in \mathbb{E}_X. kX \cap hY = zW$. It is called infinitely exponential-closed if this holds also with infinite intersections.

![Fig. 2. A non exponential-closed domain](image-url)

It is immediate to prove, by Lemma 2 that if $X \in \{\mathbb{N}, \mathbb{Z}\}$ then $\{X\}$ is an exponential-closed domain. However, this condition doesn’t hold in general for abstract domains. Consider the abstract domain in Figure 4. We can easily verify that $4 \cdot [0, 4] \cap 2 \cdot [6, 9] = 4 \cdot [3, 4]$ which is not a multiple of any element of the domain. In the following we assume $\mathbb{E} \in \{\mathbb{E}_X \in uco(\varphi(\mathbb{X})) | X \in \{\mathbb{N}, \mathbb{Z}\}\}$ be an exponential-closed domain.
Lemma 3. Let $a, b \in \mathbb{B}$ be atomic and $X, Y \in \mathbb{E}$.
\[
\{a\}^X \cap \{b\}^Y = \begin{cases} 
\{a\}^{X \cap Y} & \text{if } a = b \land X \cap Y \neq \emptyset \\
\{1\} & \text{if } 0 \in X \cap Y \land a \neq b \\
\emptyset & \text{otherwise}
\end{cases}
\]

Theorem 4. Let $a, b \in \mathbb{B}$ and $X, Y \in \mathbb{E}$ then \(\{a\}^X \cap \{b\}^Y = \{a\}^{kX} \cap \{b\}^{hY}\), where $k, h \in \mathbb{N}$ are such that $a = a^k$ and $b = b^h$.

As for the least upper bound operation is concerned, we consider the following Lemma.

Lemma 4. Let $a, b \in \mathbb{B}$ be atomic and $X, Y \in \mathbb{E}$. Then
\[
\{a\}^X \lor \{b\}^Y = \begin{cases} 
\{a\}^{X \lor Y} & \text{if } a = b \\
\mathbb{B} & \text{otherwise}
\end{cases}
\]

Theorem 5. Let $a, b \in \mathbb{B}$ and $X, Y \in \mathbb{E}$, if $a = a^k$ and $b = b^h$ for some $h, k \in \mathbb{N}$ then \(\{a\}^X \lor \{b\}^Y = \{a\}^{kX} \lor \{b\}^{hY}\).

6 Basic Domain Properties

At this point we have that $\mathbb{P}(\mathbb{B}, \mathbb{E})$, when $\mathbb{E}$ is an exponential-closed abstract domain, is a lattice ordered by inclusion. In the following we prove that the domains built so far don’t have infinite ascending chains when $\mathbb{E}$ is ACC, but they can have infinite descending chains. Moreover, in the following, we give a characterization of the meet-irreducible elements of $\mathbb{P}(\mathbb{B}, \mathbb{E})$.

Lemma 5. Let $X \in \{\mathbb{N}, \mathbb{Z}\}$, then $\mathbb{P}(\mathbb{B}, \{X\})$ is ACC.

Proof. For proving that this lattice is ACC it is sufficient to prove that each element has a finite number of upper bounds.

Now we prove that if $a \in \mathbb{B}$ then \((\{a\}^X)^u = \{\{b\}^X \mid a \in \{b\}^X\}\). We can see that:
\[
\{c\}^X \in (\{a\}^X)^u \iff \{a\}^X \subseteq \{c\}^X \iff \{1, a, a^2, \ldots\} \subseteq \{c\}^X \iff a \in \{c\}^X \iff \{c\}^X \in \{\{b\}^X \mid a \in \{b\}^X\}
\]

Now it’s clear that, if \(\{c\}^X \in \{\{b\}^X \mid a \in \{b\}^X\}\) and all the values are integer, then there exists $k \in \mathbb{X}$ such that $a = c^k$ and so $c | a$. Because the number of divisors of an integer is finite, it happens that $|\{(a)^X|^u\}| = |\{\{b\}^X \mid a \in \{b\}^X\}| < \omega$ and so we can say that the lattice is ACC. The same can be concluded when the values are rational for the same reasoning because it happens that the elements which are greater than a fraction $a/b$ reduced in lowest terms are all the fractions $c/d$, reduced in lowest terms, such that $c | a$ and $d | b$. This because $a \nmid b \land b \nmid a$ implies that $\forall k \in \mathbb{N} a^k \nmid b^k \land b^k \nmid a^k$. 
The following result relates meet-irreducibility of numerical powers with meet-irreducibility of their exponent sets.

**Lemma 6.** If \( a \in \mathbb{B} \) is atomic and \( X \in \mathbb{E} \) then \( \{ a \}^X \) is meet-irreducible if and only if \( X \) is meet-irreducible in \( \mathbb{E} \).

**Proposition 2.**

1. If \( \mathbb{E} \) is ACC then \( \mathbb{P}(\mathbb{B}, \mathbb{E}) \) is ACC.
2. \( \mathbb{P}(\mathbb{B}, \mathbb{E}) \) is not DCC.
3. Consider \( a \in \mathbb{B}, k, h, w \in \mathbb{N} \) and \( X, Y, Z \in \mathbb{E} \)
   \[
   \text{Mirr}(\mathbb{P}(\mathbb{B}, \mathbb{E})) = \left\{ \{ a \}^X \big| ((a = a^k \land kX = hY \cap wZ) \Rightarrow (kX = hY \lor kX = wZ)) \lor a = 0 \right\}
   \]
4. \( \mathbb{P}(\mathbb{B}, \mathbb{E}) = \mathcal{M}(\text{Mirr}(\mathbb{P}(\mathbb{B}, \mathbb{E}))) \).

In order to prove that \( \mathbb{P}(\mathbb{B}, \mathbb{E}) \), with \( \mathbb{E} \) being an exponential-closed domain, is an abstraction of \( \wp(\mathbb{B}) \) we have to find a GI between these two domains. Consider \( \alpha : \wp(\mathbb{B}) \to \mathbb{P}(\mathbb{B}, \mathbb{E}) \) and \( \gamma : \mathbb{P}(\mathbb{B}, \mathbb{E}) \to \wp(\mathbb{B}) \) defined as follows:

\[
\alpha(Y) = \bigcap \left\{ \{ a \}^X \big| X \in \mathbb{E} \land Y \subseteq \{ a \}^X \right\}
\]

\[
\gamma(\{ a \}^X) = \{ a \}^X
\]

The following lemmas prove that \( \alpha \) is well defined respectively whenever either \( \mathbb{E} \in \{ \{ \mathbb{N} \}, \{ \mathbb{Z} \} \} \) or \( \mathbb{E} \) is an infinitely exponential-closed domain.

**Lemma 7.** Consider \( Y \in \wp(\mathbb{B}), a \in \mathbb{B} \) and \( X \in \{ \mathbb{N}, \mathbb{Z} \} \), then

\[
| \{ a \}^X \big| Y \subseteq \{ a \}^X \}| < \omega
\]

**Proof.** Let \( y \in Y \), for Lemma 8 we can say \( \{ a \}^X \big| y \in \{ a \}^X \} = (\{ y \}^X)^u \) and \( | \{ a \}^X \big| y \in \{ a \}^X \}| < \omega \). Now we know that \( Y \subseteq \{ a \}^X \) implies that \( y \in \{ a \}^X \), so we can write \( \{ a \}^X \big| Y \subseteq \{ a \}^X \} \subseteq \{ a \}^X \big| y \in \{ a \}^X \} \), and because the second of these sets is finite then also the first one must be finite.

**Lemma 8.** Consider the families \( \{ a_i \}_{a_i \in \mathbb{B}} \) and \( \{ X_i \}_{X_i \in \mathbb{E}} \) then \( \bigcap_i \{ a_i \}^{X_i} = \{ a \}^{kX} \) for \( k \in \mathbb{N} \) and \( X \in \mathbb{E} \).

**Proof.** Consider \( a_i = a_i^{k_i} \) for \( k_i \in \mathbb{N} \) (if there exist in the family at least two values with a different e-atom, then the intersection is empty)

\[
\bigcap_i \{ a_i \}^{X_i} = \bigcap_i \{ a_i^{k_i} \} = \bigcap_i \{ a_i^k \big| h \in k_i X_i \} = \{ a_i^k \big| \forall i. h \in k_i X_i \} = \{ a_i^h \big| h \in kX \} = \{ a \}^{kX}
\]

because we supposed that \( \mathbb{E} \) was infinitely exponential-closed.

Therefore, \( \mathbb{P}(\mathbb{B}, \mathbb{E}) \) is a Moore family of \( \wp(\mathbb{B}) \) if \( \mathbb{E} \) is an infinitely exponential-closed domain or if it is \( \{ \mathbb{N} \} \) or \( \{ \mathbb{Z} \} \). The following theorem follows by Lemma 7 and 8.
Theorem 6. Let \( E \) be an infinitely exponential-closed domain or \( E \in \{\mathbb{N}, \mathbb{Z}\} \). Then \( (\mathbb{P}(E), \mathbb{E}, \alpha, \gamma, \phi(\mathbb{B})) \) is a Galois insertion.

Now that we have described the structure of the domain, it can be useful to consider a programming language with standard arithmetical operations on integers or rational numbers, and define their abstract interpretation as follows:

- **SUM**: \( \forall a, b \in \mathbb{B} \) and \( X, Y \in \mathbb{E} \) then \( \{a\}^X \oplus \{b\}^Y = \mathbb{B} \)
- **PRODUCT**: \( \forall a, b \in \mathbb{B} \) and \( X, Y, Z \in \mathbb{E} \) then \( \{a\}^X \odot \{b\}^Y = \{c\}^W \) where \( a_\uparrow = b_\uparrow = d \), with \( a = d^k \) and \( b = d^h \), and \( zW = \bigcap \{zW \mid kX \oplus hY \subseteq zW\} \), then \( c = d^z \); if \( a_\uparrow \neq b_\uparrow \) then \( \{a\}^X \odot \{b\}^Y = \mathbb{B} \).
- **DIVISION**: \( \forall a, b \in \mathbb{Z} \) and \( X, Y \in \mathbb{E}_\mathbb{N} \) then \( \{a\}^X \div \{b\}^Y = \mathbb{Z} \)
- **MODULE**: \( \forall a, b \in \mathbb{Z} \) and \( X, Y \in \mathbb{E}_\mathbb{N} \) then \( \{a\}^X \mod \{b\}^Y = \mathbb{Z} \)
- **EXPONENT**: \( \forall a \in \mathbb{Z}, b \in \mathbb{Z}^+ \) and \( X, Y \in \mathbb{E}_\mathbb{N} \) then \( \#(\{a\}^X, \{b\}^Y) = \{a\}^N \) if \( N \in \mathbb{E}_\mathbb{N}, \#(\{a\}^X, \{b\}^Y) = \mathbb{B} \) otherwise

Now we can prove the correctness of these operations, in particular we can prove that \( \alpha op \gamma = op^\delta \), namely that these abstract operations are the optimal abstraction of the corresponding concrete ones on the concrete domain \( \phi(\mathbb{B}) \), when they are defined.

**Proposition 3.** \( \oplus, \odot, \div, \mod, \# \) are optimal.

### 7 Exponential-Closed Domains

In the following we prove that well-known domains for program analysis of numerical integer or natural numbers are infinitely exponential closed, and therefore they can be used for approximating the properties of the exponent set. We consider Granger’s domain of congruences \( C(\mathbb{Z}) = \{aZ + b \mid a, b \in \mathbb{Z}\} \), Cousot’s domain of intervals \( \text{Int}(\mathbb{Z}) = \{[l, u] \mid l, u \in \mathbb{Z} \cup \{-\infty, +\infty\}, l \leq u \} \), and its restriction to \( \mathbb{N} \): \( \text{Int}(\mathbb{N}) \), and a domain for numerical power analysis described in Section \( \mathbb{P}(\mathbb{Z}), \mathbb{N}\). The first domain allows us to analyze numerical powers with exponents being of the form \( mZ + n \), with \( m, n \in \mathbb{Z} \). The second domain instead allows us to analyze the size of the exponent set, in \( \mathbb{N} \) or in \( \mathbb{Z} \). Finally, the third one analyzes the property of the exponent set of being a power set in \( \mathbb{P}(\mathbb{Z}, \{\mathbb{N}\}) \).

**Proposition 4.** \( C(\mathbb{Z}), \text{Int}(\mathbb{Z}), \mathbb{P}(\mathbb{Z}, \{\mathbb{N}\}) \in \text{uco}(\mathbb{Z}) \) and \( \text{Int}(\mathbb{N}) \in \text{uco}(\mathbb{N}) \) are infinitely exponential-closed domains.

In the following we consider some examples of intersection and least upper bound with different exponent domains. As for the intervals are concerned, note that if we consider \( X \in \{\mathbb{N}, \mathbb{Z}\} \) and \( a, b, c \in X \) then \( a\{b, c\} = aX \cap [ab, ac] \), namely to multiply a value with an interval is equivalent to consider all the multiples of the same value inside the interval.

**Example 4.** Consider \( \mathbb{B} = \mathbb{Z}, \mathbb{E} = \text{Int}(\mathbb{N}) \) and \( a, b \in \mathbb{B} \). We have to determine \( c \in \mathbb{B} \) such that \( \{c\}^X = \{a\}^{[2,10]} \cap \{b\}^{[4,15]} \) where \( a = 144 = 2^4 \cdot 3^2 \) and \( b = \ldots \)
1728 = 2^6 \cdot 3^3. We have that \( k = \gcd(2, 4) = 2 \) so \( a_1 = 2^2 \cdot 3 = 12 \) Moreover \( h = \gcd(6, 3) = 3 \) so \( b_1 = 12 \), hence the two values have the same e-atom and the intersection exists. We can easily see that \( a = 12^2 \), \( b = 12^3 \). Now we can find the intersection as \( \{12\}^{2N \cap [14, 20]} \cap \{12\}^{3N \cap [12, 45]} = \{12\}^{5N \cap [12, 20]} \). At this point we can conclude that \( c = 12^6 \) and \( X = [2, 3] \), namely \( \{a\}^{[2, 10]} \cap \{b\}^{[4, 15]} = \{c\}^{[2, 3]} \).

It’s clear that if the exponent sets of the sets of powers represented by their e-atoms, are disjoint then, independently from their value, the intersection must be empty. Consider \( \{8\}^{[5, 10]} \cap \{16\}^{[8, 15]} = \{2\}^{3N \cap [15, 30]} \cap \{2\}^{4N \cap [32, 60]} = \emptyset \) because \( [15, 30] \cap [32, 60] = \emptyset \) (Fig. 3).

**Example 5.** Consider \( \mathbb{B} = \mathbb{Q} \) and \( \mathbb{E} = C(\mathbb{Z}) \). Let \( p = \frac{5^4}{54^2} \) and \( q = \frac{54^3}{5^6} \). We want to find \( \{c\}^X = \{p\}^{5Z-1} \cap \{q\}^{5Z+2} \). In order to have all powers represented by values greater than one, note that \( \left\{ \frac{5^4}{54^2} \right\}^{5Z-1} = \left\{ \frac{54^2}{5^4} \right\}^{-5Z+1} = \left\{ \frac{54^2}{5^4} \right\}^{5Z+1} \).

Now we can see that \( k = \gcd(4, 2) = 2 \) and \( h = \gcd(6, 3) = 3 \), then \( p_1 = \frac{54^3}{5^6} \). Now we can find the intersection of the two sets as \( \{p_1\}^{4Z+2} \cap \{p_1\}^{15Z+6} = \{p_1\}^{600Z+66} \) because \( 60Z + 66 = 4Z + 2 \cap 15Z + 6 \) (remind that \( k_1Z + h_1 \cap k_2Z + h_2 = x + \text{lcm}(k_1, k_2)Z \) and \( x \in k_1Z + h_1 \cap k_2Z + h_2 \)). Hence \( c = 12^6 \) and \( X = 10Z + 11 \).

**Example 6.** Consider \( a, b \in \mathbb{B} = \mathbb{Q} \) and \( \mathbb{E} = Int(\mathbb{Z}) \). We have to find \( \{c\}^X = \{1/32\}^{[2, 10]} \cap \{8\}^{[20, 25]} \). First of all we modify the representation of the two sets.
in order to obtain the notation used in Theorem 3. We note that \( \{1/32\}\^{[2,10]} = \{32\}\^{[-10,-2]} = \{2\}^{32}\cap[[-50,-10]} \), similarly \( \{8\}\^{[20,25]} = \{2\}^{32}\cap[60,75] \). Then we see that the common e-atom of the two numbers is 2. Then \( \{1/32\}\^{[2,10]} \lor \{8\}\^{[20,25]} = \{2\}^{52}\cap[[-50,-10]} \lor \{2\}^{32}\cap[60,75] = \{2\}^{50,75} \) because we have that \([[-50,75] = 5Z \cap [-50, -10] \lor 3Z \cap [60,75] \). We can conclude that the least upper bound is \( \{2\}^{[-50,75]} \) (Fig. 4).

8 Numerical Power Program Analysis

In this section we apply the numerical power domains to the static analysis of programs by abstract interpretation. The following example shows the static analysis of numerical powers in \( \mathbb{P}(\mathbb{Z}, \{\mathbb{N}\}) \) of a simple program fragment.

Example 7. Consider the following program fragment:

\[
\begin{align*}
n &:= 1; \quad m := 4; \\
\pi_1 &:
\quad \text{while } n \leq 10000 \text{ do} \\
\pi_2 &:
\quad m := \exp(n, m); \\
\pi_3 &:
\quad n := 2 \ast n \ast m \\
\pi_4 &:
\quad \text{endw}
\end{align*}
\]

The concrete semantics \( S : \pi \rightarrow (\text{Var} \rightarrow \wp(\mathbb{Z})) \) is a function associating with each program point \( \pi \) a concrete state \( \sigma \in \text{Var} \rightarrow \wp(\mathbb{Z}) \). The following recursive equation defines the semantics of program point \( \pi_2 \), where \( \sqcup \) is the point-wise extension to functions of set union:

\[
S_{\pi_2} = S_{\pi_1} \sqcup S_{\pi_2} \left[ m \mapsto \exp(S_{\pi_2}(n), S_{\pi_3}(m)) \quad n \mapsto 2 \ast S_{\pi_2}(n) \ast \exp(S_{\pi_2}(n), S_{\pi_2}(m)) \right] \tag{1}
\]

The abstract semantics \( S^\sharp : \pi \rightarrow (\text{Var} \rightarrow \mathbb{P}(\mathbb{Z}, \{\mathbb{N}\})) \) is the least fix-point of the equation

\[
S^\sharp_{\pi_2} = S^\sharp_{\pi_1} \lor S^\sharp_{\pi_2} \left[ m \mapsto \exists(S^\sharp_{\pi_2}(n), S^\sharp_{\pi_2}(m)) \quad n \mapsto \{2\}^N \sqcap S^\sharp_{\pi_2}(n) \sqcap \exists(S^\sharp_{\pi_2}(n), S^\sharp_{\pi_2}(m)) \right] \tag{2}
\]

The solution is obtained in three steps as follows:

\[
\begin{align*}
S^\sharp_{\pi_2}(\emptyset) &= \{m \mapsto \{4\}^N, n \mapsto \{1\}^N\} \\
S^\sharp_{\pi_2}(S^\sharp_{\pi_2}(\emptyset)) &= \{m \mapsto \{4\}^N, n \mapsto \{2\}^N\} \\
S^\sharp_{\pi_2}(S^\sharp_{\pi_2}(S^\sharp_{\pi_2}(\emptyset))) &= \{m \mapsto \{2\}^N, n \mapsto \{2\}^N\} \text{ (fix-point)}
\end{align*}
\]

As for variable \( m \) is concerned: \( \exists(\{2\}^N, \{4\}^N) = \{2\}^N \) and \( \{4\}^N \lor \{2\}^N = \{2\}^N \).
It is worth noting that most abstract operations, in particular integer division, loose precision due to the unknown bound of values in numerical powers. This can be overcome by combining \( \mathbb{P}(\mathbb{Z}, \{\mathbb{N}\}) \) with the interval domain \( \text{Int}(\mathbb{Z}) \). We consider the reduced product domain \( \mathbb{P}_I(\mathbb{Z}, \{\mathbb{N}\}) = \mathbb{P}(\mathbb{Z}, \{\mathbb{N}\}) \cap \text{Int}(\mathbb{Z}) \). In this case, the abstract object \( \langle \{a\}^N, [l, u] \rangle \in \mathbb{P}_I(\mathbb{Z}, \{\mathbb{N}\}) \) represents the concrete set of integers \( \{x \mid l \leq x = a^N \leq u, n \in \mathbb{N}\} \). For instance, integer division can be improved as follows:

\[
\langle \{a\}^N, [l, u] \rangle \div \langle \{b\}^N, [l', u'] \rangle = \begin{cases} 
\{\{c\}^N, \lceil \frac{l}{u'} \rceil, \lfloor \frac{u}{b} \rfloor \} & \text{if } u' \leq l, c^k = a \text{ and } c^h = b \\
\langle \mathbb{Z}, \lceil \frac{l}{u'} \rceil, \lfloor \frac{u}{b} \rfloor \rangle & \text{otherwise}
\end{cases}
\]

**Example 8.** Consider the Collatz program in Sec. 11 and the abstract operations: \( ev : \mathbb{P}_I(\mathbb{Z}, \{\mathbb{N}\}) \rightarrow \{\text{tt}, \text{ff}, \top\} \) and \( \phi : \{\text{tt}, \text{ff}, \top\} \times \mathbb{P}_I(\mathbb{Z}, \{\mathbb{N}\}) \times \mathbb{P}_I(\mathbb{Z}, \{\mathbb{N}\}) \rightarrow \mathbb{P}_I(\mathbb{Z}, \{\mathbb{N}\}) \) such that:

\[
ev(\{a\}^N, [l, u]) = \begin{cases} 
\text{tt} & \text{if } a \text{ is even} \\
\text{ff} & \text{if } a \text{ is odd} \\
\top & \text{otherwise}
\end{cases}
\]

\[
\phi(x, \{a\}^N, [l, u], \{b\}^N, [l', u']) = \begin{cases} 
\{\{a\}^N, [l, u]\} & \text{if } x = \text{tt} \\
\{\{b\}^N, [l', u']\} & \text{if } x = \text{ff} \\
\{\{a\}^N \land \{b\}^N, [\min(l, l'), \max(u, u')]\} & \text{if } x = \top
\end{cases}
\]

In this case we obtain the abstract semantics for program point \( \pi_2 \) as in the following equation:

\[
S_{\pi_2}^g(n) = S_{\pi_1}^g(n) \lor [n \mapsto \phi(\ev(S_{\pi_2}^g(n)), S_{\pi_2}^g(n) \div \{\{2\}^N, [2, 2]\}, \langle \mathbb{Z}, [\infty, +\infty]\rangle) \land \langle \mathbb{Z}, [2, +\infty]\rangle)] \tag{3}
\]

It is immediate to prove that if \( S_{\pi_1}^g(n) = \{\{a\}^N, [l, u]\} \) with \( a \in \{2\}^N \) and \( l \geq 2 \), then \( S_{\pi_2}^g(n) = \{\{2\}^N, [2, u]\} \) is a fix-point of Eq. 3.

### 8.1 Static Analysis of Randomized \( ccp \) Programs

Rational power analysis is particularly appropriate for approximating an invariant concerning the probability of randomized programs. In particular the abstract objects in \( \mathbb{P}(\mathbb{Q}, \mathbb{E}_N) \) are suitable for approximating the probability component of computations involving randomized choices. This because the accumulated probability is usually obtained as the product of the step-by-step probability of each transition, and the product operation is extremely precise in \( \mathbb{P}(\mathbb{Q}, \mathbb{E}_N) \), with \( \mathbb{E}_N \in \{\{\mathbb{N}\}, \text{Int}(\mathbb{N})\} \). Moreover the abstract domain \( \mathbb{E}_N \) may characterize the property of exponents of the computed probability. The following proposition specifies the abstract product operation in the two cases described above.

**Proposition 5.** Consider \( \mathbb{P}(\mathbb{Q}, \mathbb{E}_N) \) with \( \mathbb{E}_N \in \{\{\mathbb{N}\}, \text{Int}(\mathbb{N})\} \), \( a, b \in \mathbb{Q}, a_1 = b_1 = d, a = a^k, b = d^h \) for some \( h, k \in \mathbb{N} \), and \( g = \gcd(k, h) \). Then
In order to model the situation described above as a static program analysis problem, we consider a probabilistic version of a concurrent constraint calculus \( pccp \) \[10\]. The syntax and operational semantics of \( pccp \) has been studied in \[17\] as in Table 1. The ask-tell paradigm, which is the basis of \( ccp \) languages, is based on the notion of blocking ask: A process is suspended when the store does not entail the ask constraint and it remains suspended until the store entails it. A constraint system represents the basic algebraic notion behind \( ccp \) \[10\]. Informally, we have an enumerable set \( D \) of elementary assertions and a finite entailment relation \( \vdash \subseteq \wp_f(D \times D) \). A constraint system is \( \Sigma \) defined \( \langle \wp(D), \vdash, \exists, d_{x,y} \rangle / \sim, \) which is a complete \( \omega \)-algebraic lattice, where \( X \sim Y \) iff \( (X)\sim = (Y)\sim \), being \( (X)\sim \) the entailment closure of a set of assertions \( X \). In order to treat the hiding operator of the language, a family of unary operations \( \exists_x \) called cylindrifications is introduced \[11\]. Diagonal elements \( d_{x,y}, \) i.e. equational constraints between variables, are considered as a way to provide parameter passing. The semantics is as usual defined by a transition system \( \longrightarrow \subseteq Conf \times \mathbb{Q} \times Conf \), where \( C \longrightarrow p C \) indicates that the transition from \( C \) to \( C' \) holds with probability \( p \in [0,1] \) and \( Conf = Agent \times \Sigma \). Note that in rule \( R2 \) an agent \( A_j \) is enabled when the store entails the ask constraint. The resulting probability \( \tilde{p}_j \) is normalized according to the enabled agents: \( \tilde{p}_j = p_j / (\sum_{c_i} p_i) \). The relational I/O semantics of a \( pccp \) program \( P \) is then defined as the following function:

\[
\wp[D.A] \defeq \lambda c. \left\{ \bigcup_i d_i, \prod_i p_i \right\} \left| \begin{array}{l}
d_0 = c \land A_0 = A \\
\langle A_i, d_i \rangle \longrightarrow p_i \langle A_{i+1}, d_{i+1} \rangle
\end{array} \right\}
\]
The abstract semantics on the GI \((P(\mathbb{Q}, \{\mathbb{N}\}), \alpha, \gamma, \wp(\mathbb{Q}))\) is defined as follows:

\[
O^a[D.A] \overset{\text{def}}{=} \lambda c. \left\{ \langle \bigcup_i d_i, \bigcap_i \alpha(\{p_i\}) \rangle \left| d_0 = c \land A_0 = A \langle A_i, d_i \rangle \rightarrow_{p_i} \langle A_{i+1}, d_{i+1} \rangle \right. \right\}
\]

The following soundness result, which relates concrete and abstract probabilities, is immediate by abstract interpretation.

**Theorem 7.** If \(\langle c, p \rangle \in O[P](d)\) then there exists \(\langle c, p^* \rangle \in O^a[P](d)\) such that \(p \in \gamma(p^*)\).

### 8.2 Analyzing Randomized Sources

An important feature of Shannon’s information theory is that the measure of information (i.e. entropy) determines the saving in transmission time that is possible by proper encoding due to statistics of the message source [20]. In order to understand this important result, let us fix a source alphabet \(S\) and an encoding alphabet \(E\), with a one-to-one function \(\mathcal{H} : S \rightarrow E^*\) which returns the encoding of each symbol in \(S\). Suppose that each symbol in \(s \in S\) is provided with a corresponding probability \(p_s \in [0, 1]\). The average length of the code \(\mathcal{H}(S)\) is

\[
\mathcal{L}(\mathcal{H}) = \sum_{s \in S} p_s \cdot |\mathcal{H}(s)|
\]

where \(|\cdot|\) indicates the length of a sequence. Usually, \(\mathcal{L}(\mathcal{H}) \geq H_S\), where \(H_S\) is measure of the information rate (or entropy) of the source: \(H_S = \sum_i p_i \log |E| 1/p_i\). The encoding of information is optimal when \(\mathcal{L}(\mathcal{H}) = H_S\) holds. The question is: *Can we statically analyze a randomized source in order to determine how to optimally encode the information from that source?* A basic result in standard information theory says that \(\mathcal{L}(\mathcal{H}) = H_S\) holds when for any \(s \in S\): \(p_s = (1/|E|)^n\) with \(n \in \mathbb{N}\) [20]. In our setting, where the randomized source is a program with probabilistic choice [18], this analysis corresponds precisely to statically analyze the probability of the objects produced by the program (see [10] for a general framework for probabilistic program analysis). When this probability lies in a rational power of type \(\{1/a\}^\mathbb{N}\), then any encoding alphabet \(|E| = a\) can determine the optimal encoding of the source. The result of the analysis is useful in finding, effectively, an optimal code for the source given. For example we can use Huffman’s algorithm [12] which takes the source symbols with their probabilities, the cardinality \(s\) of the source and the cardinality \(t\) of the code alphabet, and returns an optimal code for the symbols of the source. We can observe that the Huffman’s algorithm with a fixed encoding alphabet, gives always the optimal code for the particular situation. If we don’t fix the size of the encoding alphabet then, with the analysis described above, we can find the best one for this alphabet as the value of the entropy of the source when all the probabilities are power of the same value. In this way, if we assume \(t = a\) where \(a\) is the result of the static analysis of the source, then this code has the least possible average length.
Example 9. Consider a randomized counting algorithm $P$ expressed in pccp [17].

$$
P : \text{nat}(x) : - \quad \text{true}|\frac{1}{2} \rightarrow \text{tell}(x = 0) \\
\quad \Box \text{true}|\frac{1}{2} \rightarrow \exists y (\text{tell}(x = s(y))||\text{nat}(y)).\text{nat}(x)
$$

The program $P$ generates an infinite sequence of natural numbers $\mathcal{O}[P](\text{true}) = \{\langle x = s^n(0), \frac{1}{2^{n+1}} \rangle \mid n \geq 0\}$ with decreasing probability. This information can be automatically derived by abstract interpretation in $\mathcal{P}(\mathbb{Q}, \{\mathbb{N}\})$. The concrete domain is $\wp(\mathbb{N} \times \mathbb{Q})$. It is immediate to derive the following information by abstract interpretation of $\mathcal{O}[P]$ in a product domain $\mathcal{T} \times \mathcal{P}(\mathbb{Q}, \{\mathbb{N}\})$, where $\mathcal{T} = \{\text{nat}\}$ captures basic type information. In this case, the approximated semantics is $\mathcal{O}^\sharp[P](\text{true}) = \{\text{nat}, \{1/2\}^\mathbb{N}\}$.

9 Conclusions

We have built a family of abstract domains, with a common structure, which are useful for analyzing numerical (integer or rational) powers. These domains are parametric on the abstraction of the exponent set. Our results so far provides the possibility to design new abstract domains for numerical power analysis by plugging suitable abstractions of the exponent set. This construction can be further generalized by considering the possible algebraic structure of the base set $\mathbb{B}$. It is well-known that euclidean rings provide the appropriate algebraic structure that allows prime factorization of its elements. This is a key point in our construction of $\mathcal{P}(\mathbb{B}, \mathcal{E})$. However, it is worth noting that $\mathbb{N}$ is not a ring, even though it allows the prime factorization. Therefore we believe that it is possible to find a more abstract algebraic structure for $\mathbb{B}$ such that $\mathcal{P}(\mathbb{B}, \mathcal{E})$ is an abstraction of $\wp(\mathbb{B})$. This would be particularly important in order to generalize the numerical power analysis to the power of other, possibly non-numerical, objects (e.g., polynomials).

Acknowledgements

I thank very much Roberto Giacobazzi for all the discussions and all the time that he spent for helping me in the creation of this work. I also thank the anonymous referees for their useful comments on the first version of this paper.

References


Run-Time Bytecode Specialization
A Portable Approach to Generating Optimized Specialized Code

Hidehiko Masuhara and Akinori Yonezawa

1 Department of Graphics and Computer Science
Graduate School of Arts and Sciences, University of Tokyo
masuhara@acm.org

2 Department of Information Science, University of Tokyo
yonezawa@is.s.u-tokyo.ac.jp

Abstract. This paper proposes a run-time bytecode specialization (BCS) technique that analyzes programs and generates specialized programs at run-time in an intermediate language. By using an intermediate language for code generation, a back-end system can optimize the specialized programs after specialization. As the intermediate language, the system uses Java virtual machine language (JVML), which allows the system to easily achieve practical portability and to use sophisticated just-in-time compilers as its back-end. The binding-time analysis algorithm, which is based on a type system, covers a non-object-oriented subset of JVML. A specializer, which generates programs on a per-instruction basis, can perform method inlining at run-time. The performance measurement showed that a non-trivial application program specialized at run-time by BCS runs approximately 3–4 times faster than the unspecialized one. Despite the large amount of overheads at JIT compilation of specialized code, we observed that the overall performance of the application can be improved.

1 Introduction

Given a generic program and the values of some parameters, partial evaluation techniques generate a specialized program with respect to the values of those parameters [11,17]. Most of those techniques have been studied as source-to-source transformation systems; i.e., they analyze programs in a high-level language and generate specialized programs in the same language. They have been successful in the optimization of various programs, such as interpreters, scientific application programs, and graphical application programs [4,13,20].

Run-time specialization (RTS) techniques [10,12,13,22] efficiently perform partial evaluation at run-time (1) by constructing a specializer (or a generating extension) for each source program at compile-time and (2) by directly generating native machine code at run-time. The drastically improved specialization speed enables programs to be specialized by using values that are computed at run-time, which means that RTS provides more specialization opportunities...
than compile-time specialization. Several studies reported that RTS can improve performance of programs for numerical computation [18,22], an operating system kernel [25], an interpreter of a simple language [18], etc.

One of the problems of RTS systems is a trade-off between efficiency of specialization and efficiency of specialized code. For example, Tempo generates specialized programs by merely copying pre-compiled native machine code. The performance of the generated code is 20% slower than that is generated by compile-time specialization on average [22]. Of course, we could optimize specialized programs at run-time by optimizing generated code after specialization. It however makes amortization more difficult.

In this paper, we describe an alternative approach called run-time bytecode specialization (BCS), which is an automatic bytecode-to-bytecode transformation system. The characteristics of our approach are: (1) the system directly analyzes program and constructs specializers in a bytecode language; and (2) the specializer generates programs in the bytecode language, which makes it possible to apply optimizations after specialization by using just-in-time (JIT) compilation techniques.

As the bytecode language, we choose the Java virtual machine language (JVML) [19], which provides us practical portability. The system can use existing compilers as its front-end, and widely available Java virtual machines, which may include sophisticated JIT compilers, as its back-end. The analysis of JVML programs is based on a type system derived from the one for JVML [27]. A specializer can be basically constructed from the result of the analysis, and can perform method inlining at run-time.

Thus far, we have developed our prototype system for a non-object-oriented subset of JVML; the system support only primitive types, arrays, and static methods. Although the system does not yet support important language features in Java, such as objects and virtual methods, it has sufficient functionality to demonstrate fundamental costs in our approach, such as efficiency of specialized code and overheads of specialization and JIT compilation.

The rest of the paper is organized as follows. Section 2 overviews existing RTS techniques and their problems. BCS is described in Section 3. Section 4 presents the performance measurement of our current implementation. Section 5 discusses related studies. Section 6 concludes the paper.

2 Run-Time Specialization

2.1 Program Specialization

An offline partial evaluator processes programs in two phases: binding-time analysis (BTA) and specialization. BTA takes a program and a list of the binding-times of arguments of a method in the program and returns an annotated pro-

---

1 In RTS systems, a specialization process of a procedure is amortized if the amount of reduced execution time of the procedure becomes larger than the time elapsed for the specialization process.
gram in which every sub-expression is associated with binding-time. The binding-
time of an expression is *static* if the value can be computed at specialization time
or *dynamic* if the value is to be computed at execution time. For example, when
BTA receives

```java
class Power
{
    static int power(int x, int n)
    {
        if (n==0) return 1;
        else return x*power(x,n-1);
    }
}
```

with list [dynamic, static] as the binding-times of x and n, it adds static anno-
tations to the *if* and *return* statements, to the expressions n==0 and n-1, and
to the call to *power*. The remaining expressions, namely the constant 1 in the
‘then’ branch, the variable x, and the multiplication, are annotated as dynamic.

In the specialization phase, the annotated program is executed with the val-
ues of the static parameters, and a specialized program is returned as a result.
The execution rules for static expressions are the same as the ordinary ones. The
rule for the dynamic expressions is to return the expression itself. For example,
execution of annotated *power* with argument 3 for static parameter n proceeds
as follows: it tests “n==0”, then selects the ‘else’ branch, computes n-1, and rec-
ursively executes *power* with 2 (i.e., the current value of n-1) as an argument.
It eventually receives the result of the recursive call, which should be “x*x*1”,
and finally returns “x*x*x*1” by appending the received result to “x*”.

### 2.2 Overview

*Run-time specialization* techniques efficiently specialize programs by generating
specialized programs at machine code level [10,12,15,18,22,29].

Given a source program and binding-time information, an RTS system effi-
ciently generates specialized programs at run-time in the following way. It first
performs BTA on the source program, similar to compile-time specialization
systems. It then compiles dynamic expressions into fragments of machine code,
called *templates*. It also constructs a *specializer* that has the static expressions
in the source program and operations for copying corresponding templates into
memory in place of the dynamic expressions. At run-time, when a specializer
is executed with a static input, it executes the static expressions, and directly
generates a specialized program at machine code level.

### 2.3 Problems

**Efficiency.** There is a trade-off between efficiency of specialization processes
and efficiency of specialized code in RTS systems.

A program that is specialized by an RTS system is usually slower than that
specialized by a compile-time specialization system. This is because RTS systems
rarely apply optimizations, such as instruction scheduling and register allocation,
to the specialized code, for the sake of efficient specialization. Furthermore, pro-
grams that have a number of method invocations (or function calls) would be
much slower since method inlining is not performed in several RTS systems.

For example, Noël, et al. showed that the run-time specialized programs
have 20% overheads over the compile-time specialized ones on average, in their
study on Tempo\cite{_tempo}. As we will see in Section 4, the overheads in the run-time
specialized program can overwhelm the speedup obtained by specialization; i.e.,
the specialized program become slower than the original program.

If an RTS system performed optimizations at run-time, specialized programs
would become faster. In fact, there are several systems that optimizes specialized
code at run-time\cite{open_jit,retargetable}. However, the time spent for the optimization processes
makes amortization more difficult.

Consequently, an RTS system that can flexibly balance a degree of opti-
mization of specialized code and time for generating specialized code would be
beneficial.

**Portability.** In order to directly generate machine code, RTS systems often de-
pend on the target machine architecture. A typical RTS system includes its own
compiler from source code (usually in a high-level language) to native machine
code.

Several techniques have been proposed to overcome the problem. For exam-
ple, Tempo uses standard C compilers for creating templates\cite{tempo,standard_c}. C, which
is a language with dynamic code generation mechanisms, generates specialized
code in *retargetable* virtual machine languages called vcode and icode\cite{retargetable}.

## 3 Run-Time Bytecode Specialization

### 3.1 Overview

Our proposed *run-time bytecode specialization* (BCS) technique uses a virtual
machine (bytecode) language as its source and target languages. It takes a byte-
code program as its input, and constructs a specializer in the same bytecode
language. At run-time, the specializer, which runs on a virtual machine, gener-
ates specialized programs in the same bytecode language. As a virtual machine
language, we choose the Java virtual machine language (JVML)\cite{jvml}.

We aim to solve the problems in the previous section in the following ways:

**Efficiency.** Instead of directly generating specialized code in a native machine
language, BCS generates it in an intermediate (bytecode) language. When
the system is running on a JVM with a JIT compiler, the specialized code
is optimized into a native machine language before execution. We also plan
to control the quality of specialized code and the speed of JIT compilation
processes by integrating our system with JVMs that have interfaces to its
JIT compilers, such as OpenJIT\cite{open_jit}.

Another functionality of specializers in BCS is that they can perform method
inlining at run-time. Although the specialized code with method inlining has
a certain amount of overheads for saving/restoring local variables at bytecode level, a JIT compiler can remove most of them according to our experiments.

**Portability.** As is shown in a previous run-time code generation system [9,24], code generation at the virtual machine level can improve portability. Current BCS system generates specialized code in the standard JVML; the generated code can be executed on JVMs that are a widely available to various platforms.

The input to the BCS system is a JVML program. This means that the system does not depend on the syntax of high-level languages. Instead, run-time specialization can be applied to any language for which there exists a compiler into JVML. In fact, there are several compilers from various high-level languages to JVML [3,6, etc.], which would be used as a front-end when we extended our system to support the full set of JVML.

As shown in Figure 1, a compiler first translates a source program written in a high-level language (e.g., Java) into JVML bytecode. The compiled program is annotated by using our BTA algorithm. From the annotated program, a specializer for generating the dynamic instructions is constructed. At run-time, the specializer takes the values for the static parameters and generates a specialized program in bytecode by writing the dynamic instructions in an array. Finally, the JVM’s class loader and the JIT compiler translate the bytecode specialized program into machine code, which can be executed as a method in the Java language.

In the following subsections, we present the outline of each process in BCS briefly. More detailed description can be found in the other literature [21].

### 3.2 Source and Target Language

As mentioned, our source and target language is JVML, which is a stack-machine language with local variables and instructions for manipulating objects. Currently, a subset of the JVML instructions is supported. Restrictions are:

- Only primitive types and array types are supported. (*i.e.*, objects are not supported yet).
Method int Power.power(int, int)
0 iload 1 // push n
1 ifne 4 // go to 4 if n ≠ 0
2 icontst 1 // (case n = 0) push 1
3 ireturn // return 1
4 iload 0 // (case n ≠ 0) push x
5 iload 0 // push x as arg. #0
6 iload 1 // push n
7 icontst 1 // push 1
8 isub // compute (n – 1) as arg. #1
9 invokestatic int Power.power(int, int) // call method
10 imul // compute x × (return value)
11 ireturn // return x × (return value)

Fig. 2. Method power in JVML.

- All methods must be class methods (i.e., methods are declared static).
- Subroutines (jsr and ret), exceptions, and multi-threading are not supported.

Figure 2 shows the result of compiling method power (Section 2.1) into JVML. A method invocation creates a frame that holds an operand stack and local variables. An instruction first pops zero or more values off the stack, performs computation, and pushes zero or one value onto the stack.

The icontst n instruction pushes a constant n onto the stack. The isub (or imul) instruction pops two values off the stack and pushes the difference (or multiple) of them onto the stack. The iload x instruction pushes the current value of local variable x onto the stack. The istore x instruction pops a value off the stack and assigns it to local variable x. The ifne L instruction pops a value off the stack and jumps to address L in the current method if the value is not zero. The invokestatic t0 m(t1, ..., tn) instruction invokes method m with the first n values on the stack as arguments. The invokestatic instruction (1) pops n values off the stack, (2) saves the current frame and program counter, (3) assigns the popped values into variables 0, ..., (n − 1) in a newly allocated frame, and (4) jumps to the first address of method m. The ireturn instruction (1) pops a value off the stack, (2) disposes of the current frame and restores the saved one, (3) pushes the value on the restored stack, and (4) jumps to the next address of the saved program counter. The caller uses the value at the top of the stack as a returned value.

3.3 Binding-Time Analysis

Strategy Our BTA algorithm is a flow sensitive and monovariant (context insensitive) analysis for the subset of JVML based on a type system. From the viewpoint of BTA, the subset of JVML is mostly similar to high-level imperative languages such as C. Therefore, the analysis should be careful about the following respects, unlike the analyses for functional languages:
Since compilers may assign different variables to an operand-stack entry or a local variable, the analysis should be flow sensitive \[16\]; i.e., it should allow an operand-stack entry or a local variable to have a different binding-time at each program point in a method.

As JVML is an unstructured language (i.e., it has a ‘goto’ instruction), merge points of a conditional jump and loops are implicit. The algorithm therefore has to somehow infer this information.

The BTA algorithm is based on a type system, following the algorithms used for functional languages \[14\]. As the type system, we use a modified version of a type system of JVML proposed by Stata and Abadi \[27\]. The algorithm, which is described in the other literature \[21\], consists of the following steps:

1. In a given program, for each address in each method, it first gives three type variables to an operand-stack, to a frame of local variables, and to an instruction at the address. By giving different type variables to local variables at each address, the system achieves flow sensitivity, as well as the original Stata and Abadi’s system.
2. It then applies typing rules to each instruction of a method, and generates constraints among the type variables.
3. It also generates additional constraints that treat non-local side-effects under dynamic control \[17\] chapter 11] by using the result of a flow analysis.
4. It finally computes a minimal set of assignments to type variables that satisfies all the generated constraints.

**Example** Figure 3 shows an example BTA result of `power` when the binding-times of `x` and `n` are dynamic and static, respectively \[3\]. The binding-time of an instruction, which is displayed in the $B$ column, is either $S$ (static) or $D$ (dynamic). The binding-time of a stack, which is displayed in the $T$ column, is written as $\tau_1 \cdot \tau_2 \cdots \cdot \tau_n \cdot \epsilon$ (a stack with $n$ values whose types are $\tau_1$, $\tau_2$, ..., from the top value). The binding-time of a frame of local variables, which is displayed in the $F$ column, is denoted as $\emptyset$ (an empty frame) or $[i_k \mapsto \tau_k]$ (a frame whose local variable $i_k$ has type $\tau_k$). Note that the domains of the frame types ‘shrink’ along the execution paths. This is because our BTA rules generate constraints on only types of live local variables, and the types of unused ones do not appear in the result.

The BTA result is effectively the same as that of the source-level BTA; i.e., instructions that correspond to a static or dynamic expression at source-level have the static or dynamic types, respectively.

\[2\] They design their type system for formalizing the JVM’s verification rules in terms of subroutines (jsr and ret). Here, our current analysis merely uses the style of their formalization, and omits complicated rules for subroutines.

\[3\] The instruction sequence was slightly modified from Figure 2, so that any conditional jump has merge points within the method. A preprocessor inserts a unique ireturn instruction at the end of the method, and replace all ireturn instructions with goto instructions to the inserted ireturn instruction.
Run-Time Bytecode Specialization

<table>
<thead>
<tr>
<th>instruction</th>
<th>( B )</th>
<th>( T )</th>
<th>( F )</th>
</tr>
</thead>
<tbody>
<tr>
<td>iload 1</td>
<td>( S )</td>
<td>( \epsilon )</td>
<td>( 0 \mapsto D, 1 \mapsto S )</td>
</tr>
<tr>
<td>ifne L2</td>
<td>( S )</td>
<td>( S \cdot \epsilon )</td>
<td>( 0 \mapsto D, 1 \mapsto S )</td>
</tr>
<tr>
<td>( L1 : ) icnst 1</td>
<td>( S )</td>
<td>( \epsilon )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>goto L0</td>
<td>( S )</td>
<td>( D \cdot \epsilon )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( L2 : ) iload 0</td>
<td>( D )</td>
<td>( \epsilon )</td>
<td>( 0 \mapsto D, 1 \mapsto S )</td>
</tr>
<tr>
<td>iload 0</td>
<td>( D )</td>
<td>( D \cdot \epsilon )</td>
<td>( 0 \mapsto D, 1 \mapsto S )</td>
</tr>
<tr>
<td>iload 1</td>
<td>( S )</td>
<td>( D \cdot D \cdot \epsilon )</td>
<td>( 1 \mapsto S )</td>
</tr>
<tr>
<td>icnst 1</td>
<td>( S )</td>
<td>( S \cdot D \cdot D \cdot \epsilon )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>isub</td>
<td>( S )</td>
<td>( S \cdot S \cdot D \cdot D \cdot \epsilon )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>invokestatic</td>
<td>( S )</td>
<td>( S \cdot D \cdot D \cdot \epsilon )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( \text{int Power} \cdot \text{power}(\text{int},\text{int}) )</td>
<td>( D )</td>
<td>( D \cdot D \cdot \epsilon )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>goto L0</td>
<td>( S )</td>
<td>( D \cdot \epsilon )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( L0 : ) ireturn</td>
<td>( S )</td>
<td>( D \cdot \epsilon )</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>

Fig. 3. BTA result of \texttt{power}.

3.4 Specializer Construction

From an original program and a result of BTA, a specializer is constructed in “pure” JVML. It generates specialized code on a per-instruction basis at run-time\(^9\). For each dynamic instruction in the original program, the specializer has a sequence of instructions that writes the bytecode of the instruction into an array. The specializer also performs method inlining by successively running specializers of a method caller and callee, and by inserting a sequence of instructions that saves and restores local variables appropriately.

Here, we describe the construction of a specializer by using pseudo-instructions. Note that those pseudo-instructions are used only for explanation, and they are replaced with sequences of pure JVML instructions in the actual specializer. The specializer is executable as a Java method.

The extended JVML for defining specializers contains the JVML instructions and pseudo-instructions, namely, \( \text{GEN} \) \( \text{instruction} \), \( \text{LIFT} \), \( \text{LABEL} \) \( L \), \( \text{SAVE} \) \( n \) \([x_0, \ldots]\), \( \text{RESTORE} \), and \( \text{INVOKEGEN} \) \( m \) \([x_0, \ldots]\), where \( \text{instruction} \) is a standard JVML instruction. Figure 4 shows an example definition of specializer \texttt{power_gen} with pseudo-instructions, constructed from method \texttt{power}. A specializer is constructed by translating each annotated instruction as follows.

- Static instruction \( i \) becomes instruction \( i \) of the specializer.
- Dynamic instruction \( i \) is translated into pseudo-instruction \( \text{GEN} \) \( i \). When \( \text{GEN} \) \( i \) is executed at specialization time, the binary representation of \( i \) is written in the last position of an array where specialized code is stored.
- When an instruction has a different binding-time than that of the value pushed or popped by the instruction, pseudo-instruction \( \text{LIFT} \) is inserted. More precisely, \( (1) \) when a static instruction at \( pc \) pushes a value onto the stack and \( T[pc + 1] = D \cdot \sigma \), where \( \sigma \) denotes an arbitrary stack type, \( \text{LIFT} \)
Method Power.power_gen(int)

<table>
<thead>
<tr>
<th>iload_1</th>
<th>iload_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ifne L2</td>
<td>icost_1</td>
</tr>
<tr>
<td>L1:iconst_1</td>
<td>isub</td>
</tr>
<tr>
<td>LIFT</td>
<td>INVOKEGEN Power.power_gen(int) []</td>
</tr>
<tr>
<td>goto L0</td>
<td>GEN imul</td>
</tr>
<tr>
<td>L2:GEN iload_0</td>
<td>goto L0</td>
</tr>
<tr>
<td>GEN iload_0</td>
<td>L0:return</td>
</tr>
</tbody>
</table>

Fig. 4. Specializer definition with pseudo-instructions.

is inserted after the instruction. The icost_1 at L1 in Figure 4 is an example. (2) When a dynamic instruction at pc pops a value off the stack and \(T[pc] = S \cdot \sigma\), \LIFT is inserted before the instruction. The execution of a \LIFT instruction pops value \(n\) off the stack and generates instruction “icost \(n\)” as an instruction of the specialized program.

- Static \texttt{invokestatic} \( t_0 \ m(t_1, \ldots, t_n) \) is translated into pseudo-instruction \texttt{INVOKEGEN} \( m\_gen(t_{j_1}, \ldots, t_{j_k}) \) \([x_0, x_1, \ldots]\), where \(t_{j_1}, \ldots, t_{j_k}\) are the types of static arguments, and \(x_0, x_1, \ldots\) are the dynamic local variables at the current address. When \texttt{INVOKEGEN} is executed, (1) instructions that save local variables \(x_0, x_1, \ldots\) to the stack and move values on top of the stack to the local variables are generated, (2) a specializer \(m\_gen\) is invoked, and (3) instructions that restore saved local variables \(x_0, x_1, \ldots\) are generated. The number of values moved from the stack to the local variables in (1) is the number of dynamic arguments of \(m\).

- When conditional jump \texttt{ifne} \(L\) is dynamic, the specializer has an instruction that generates \texttt{ifne}, followed by the instructions for the ‘then’ and ‘else’ branches. In other words, it generates specialized instruction sequences of both branches, one of which is selected by the dynamic condition. First, the jump instruction is translated into two pseudo-instructions: \texttt{GEN ifne} \(L\) and \texttt{SAVE} \(n\) \([x_0, x_1, \ldots]\), where \(n\) and \([x_0, x_1, \ldots]\) are the number of static values on the stack that will be popped during the execution of the ‘then’ branch and a list of static local variables that may be updated during execution of the ‘then’ branch, respectively. In addition, pseudo-instruction sequence \texttt{LABEL} \(L\); \texttt{RESTORE} is inserted at label \(L\). When \texttt{SAVE} is executed at specialization time, the top \(n\) values on the current stack and the local variables \(x_0, x_1, \ldots\) are saved. The execution of \texttt{RESTORE} resets the saved values on the stack and in the frame.

\(^4\) Since JVML is an unstructured language, construction of a generating extension whose control flow visits all the nodes in both branches is not trivial. The algorithm for constructing such a generating extension will be explained in the other literature.
Run-Time Bytecode Specialization

Method int power_2(int)  5 istore_0
0 iload_0  6 iconst_1
1 iload_0  7 imul
2 istore_0  8 imul
3 iload_0  9 ireturn
4 iload_0

Fig. 5. Specialized version of \texttt{power} with respect to 2.

3.5 Specializer Execution

The specializer definition is further translated into a Java method so that it takes (1) several parameters needed for specialization including an array \texttt{byte[]} \texttt{code} in which instructions of the specialized program are written and (2) the static arguments of the original method.

When a program uses the specializer, the following operations are performed:

- (CP creation) A ‘Constant Pool’ (CP) object that records lifted values during specialization is created.
- (specializer execution) The specializer method is invoked with static arguments and the other necessary information for specialization.
- (class finalization) From the specialized instructions written in a \texttt{byte} array and the CP object, a ClassFile image\footnote{Despite its name, a ClassFile image in our system is created as a \texttt{byte} array. No files are explicitly created for class loading.} is created.
- (class loader creation) A ClassLoader object is created\footnote{Since some JVM implementations significantly slowed down when a ClassLoader object loads a number of classes in our experiment, we create a class loader for each specialized code. Section\textsuperscript{[4,5]} shows that the time for creating of a ClassLoader object is insignificant among the overall specialization overheads.}.
- (class loading) Using the ClassLoader object, the ClassFile image is loaded into the JVM, which defines a new class with the specialized method.
- (Instance creation) An instance of the newly defined class is created. The program finally can call the specialized method via a virtual method of the instance.

Figure\textsuperscript{[4]} shows the instructions for specialized \texttt{power} with 2 as a static argument. Some instructions, such as those that load a value immediately after storing the value, are unnecessary. Those instructions arise to save/restore local variables around inlined methods.

4 Performance Measurement

4.1 An Application Program: Mandelbrot Sets Drawer

As a target of specialization, we took a non-trivial application program that interactively displays the Mandelbrot sets. The user of the program can enter the definition of a function, and the program displays the image of the Mandelbrot set that is defined by using the function. Since the function is given interactively,
the program defines an interpreter for evaluating mathematical expressions. In order to draw an image of the set, the application have to evaluate the function more than one million times. This means that run-time specialization of the interpreter with respect to a given expression could improve the performance of the drawing process.

In our performance measurements, the method `eval` and its auxiliary methods in the interpreter, which take an expression and a store, and returns the value of the expression, are specialized with respect to an expression “z*z+c”. Since current BCS implementation does not support objects, we modified the method to use arrays for representing expressions and stores.

We measured execution times of the target methods on two JVMs with different JIT compilers, namely, Sun “Classic” VM for JDK 1.2.1 with sunwjit compiler, and Sun “HotSpot” VM for JDK 1.2.2, in order to examine impacts of a JIT strategy on the specialization performance. All programs are executed on Sun Enterprise 4000 with 14 UltraSPARCs at 167MHz, 1.2GB memory, and SunOS 5.6. Execution times are measured by inserting `gethrvtime` system calls, which is called via a native method.

4.2 Performance of Specialized Method

We measured performance of three versions of the `eval` method on the above-mentioned JVMs. The first one is the ‘original’ unspecialized method. The second one is a run-time specialized (‘RTS’) method generated by the BCS system. The third one is a compile-time specialized (‘CTS’) method, which is obtained by applying Tempo after translating the original method into a C function.
Table 1. Execution times and relative speeds of eval method.

<table>
<thead>
<tr>
<th>VM</th>
<th>execution times (µsec.)</th>
<th>relative speed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>original</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$B_O$</td>
<td>$J_O$</td>
</tr>
<tr>
<td>Classic</td>
<td>6.405</td>
<td>2.513</td>
</tr>
<tr>
<td>HotSpot</td>
<td>2.774</td>
<td>245,156</td>
</tr>
<tr>
<td></td>
<td>$B_R$</td>
<td>$J_R$</td>
</tr>
<tr>
<td></td>
<td>2.555</td>
<td>1,330</td>
</tr>
<tr>
<td></td>
<td>0.691</td>
<td>146,795</td>
</tr>
<tr>
<td></td>
<td>$B_C$</td>
<td>$J_C$</td>
</tr>
<tr>
<td></td>
<td>1.792</td>
<td>0.659</td>
</tr>
<tr>
<td></td>
<td>159,688</td>
<td>4.014</td>
</tr>
<tr>
<td></td>
<td>$B_O/B_R$</td>
<td>$B_O/B_C$</td>
</tr>
<tr>
<td></td>
<td>2.841</td>
<td>2.838</td>
</tr>
<tr>
<td></td>
<td>$B_R/B_C$</td>
<td>0.999</td>
</tr>
</tbody>
</table>

Table 2. Breakdown of specialization overheads.

<table>
<thead>
<tr>
<th>process</th>
<th>VM</th>
<th>time (µsec.) (ratio)</th>
<th>VM</th>
<th>time (µsec.) (ratio)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CP creation</td>
<td>Classic</td>
<td>46.38 (1.7%)</td>
<td>HotSpot</td>
<td>95.91 (3.1%)</td>
</tr>
<tr>
<td>specializer execution</td>
<td>61.67 (2.3%)</td>
<td>194.81 (6.2%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>class finalization</td>
<td>55.77 (2.1%)</td>
<td>125.18 (4.0%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>class loader creation</td>
<td>16.68 (0.6%)</td>
<td>22.14 (0.7%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>class loading</td>
<td>1,907.33 (71.8%)</td>
<td>1,518.18 (48.5%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>instance creation</td>
<td>569.73 (21.4%)</td>
<td>1,172.96 (37.5%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>total ($S$)</td>
<td></td>
<td>2,657.57 (100.0%)</td>
<td></td>
<td>3,129.19 (100.0%)</td>
</tr>
</tbody>
</table>

Figure 6 shows the execution times of the method, which are measured by the following way. AClassLoader object in our benchmark program first loads a new class that contains the (either specialized or unspecialized) eval method. The program then measures execution time of a loop that repeatedly invokes the eval method. Note that the measured time does not include specialization process, but does include the time of JIT compilation processes because JVMs perform JIT compilation during method invocations. As a result, the curves of the graph are not linear for small iteration numbers.

We therefore estimated, for each curve, execution times of the JIT-compiled body of the method (hereafter referred to as $B$) and JIT compilation process ($J$), by using an linear approximation of the curve at large iteration numbers.

Table 1 shows the estimated execution times and relative speed of the body of the method. As we see in the $J_O$, $J_R$ and $J_C$ columns, JIT compilation processes took from one millisecond to a few hundred milliseconds, depending on the JIT compilers. As we see in the $B_O/B_R$ and $B_R/B_C$ columns, the run-time specialized code runs 3–4 times faster than the unspecialized one does, and achieves almost the same speedup factors as the compile-time specialized code does.

4.3 Specialization Overheads and Break-Even Points

Elapsed times for the specialization processes ($S$) are measured by averaging 10,000 runs. Table 2 shows the time for each sub-process, which is explained in Section 3.5. As we see, 80–90% time of the specialization process is spent for the
ones inside the JVM, namely, class loading and instance creation. We presume
that some of overheads could be removed if we integrated our system with a
JIT compiler so that the specializer directly generates specialized code in an
intermediate representation in the JIT compiler.

A break-even point (BEP) is a number of runs of a specialized program needed
to amortize the specialization cost over the execution time of the unspecialized
program. In programming systems that perform dynamic optimizations, even
unspecialized programs have to pay overheads of the optimization, namely JIT
compilation time. We therefore calculated two BEPs. The first one assumes that
the unspecialized code is already JIT compiled. In this case, a BEP, which is
calculated by the formula \((J_R + S)/(B_O - B_R)\), is approximately 1,000–72,000
runs as shown in Table 3. The second one assumes that the unspecialized code
is newly loaded, and thus pays the cost of JIT compilation during its execution.
The BCS specialized code exhibits a small BEP in this case, which is computed
by the formula \((J_R + S - J_O)/(B_O - B_R)\). Note that the benchmark application,
in order to draw an image of a given expression, executes the eval method for
much larger number of times than the BEPs. This means that BCS actually
improves the overall execution times of the application.

4.4 Comparison to a Native Code Run-Time Specialization System

In order to compare the speedup factors and specialization overheads with a run-
time native-code specialization system, we also wrote the same interpreter in C,
and specialized by using Tempo 1.19. We have tested two binding-time con-
figuations for specializing the interpreter. The one is to specialize the function
with respect to three out of five arguments (shown in the ‘3/5’ row in Table 4),
which is the same configuration to the experiment in BCS. The other is to spe-
cialize with respect to two out of five arguments (the ‘2/5’ row), in which an
array containing a return value index is set to be dynamic. The interpreter is
compiled by GCC 2.7.2 with -O2 option. All the other execution environments
are the same to the previous ones.

Table 4 shows the execution times and specialization times that are measured
by averaging ten million runs. We observe that the run-time specialized code is
slower than the compile-time specialized one in Tempo. Surprisingly, the run-
time specialized code that is specialized under the same configuration to the

---

7 We set both reentrant and post_inlining options of Tempo to true, and the
compiler options for both templates and specializers to "-O2". We also implemented
an efficient memory allocator for residual code.
Table 4. Execution and specialization times and break-even points of eval in Tempo.

<table>
<thead>
<tr>
<th># of static args.</th>
<th>execution times (µsec.)</th>
<th>relative speed</th>
<th>BEP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BO</td>
<td>BR</td>
<td>S</td>
</tr>
<tr>
<td>3/5</td>
<td>1.278</td>
<td>63.591</td>
<td>85.712</td>
</tr>
<tr>
<td>2/5</td>
<td>1.278</td>
<td>0.789</td>
<td>22.881</td>
</tr>
</tbody>
</table>

experiments in the previous subsections is even slower than the original code. We presume that this anomaly is caused by a number of array accesses whose indices are ‘lifted’ at specialization time. When we made the array to be dynamic (the ‘2/5’ row), the run-time specialized function become faster than the original one, and its break-even point is smaller than the ones in BCS.

Comparing between the execution time in BCS and the one in Tempo, we notice that compile-time specialized codes in those two systems show the similar speedup factors (BO/BC). On the other hand, the speedup factors of the run-time specialized code (BO/BR) in Tempo are worse than the one in BCS. This can be an evidence of our premise: performing optimizations after specialization could be useful to improve performance of run-time specialized code.

5 Related Work

Tempo is a compile-time and run-time specialization system for C language[22]. Tempo achieves portability by using outputs of standard C compilers to construct specializers. As the specializers simply copy templates to memory at runtime, their BEP numbers are low (3 to 87 runs in their realistic examples). On the other hand, the specializers perform no optimizations and no function inlining at run-time specialization.

DyC is another RTS system for C language[12]. The analysis and specializers can directly handle unstructured C programs. The system generates highly optimized code, by developing its own optimizing compiler for Digital Alpha 21164. It can perform optimizations at run-time specialization[2]. However, the optimizations seem to make the BEP numbers larger (around 700 to 30,000), similar to BCS.

Fabius is an RTS for pure-functional subset of ML, targeting MIPS R3000[18]. Because the source language is a pure functional language, the BTA and specializer construction in Fabius are simpler than those for imperative and unstructured languages. Similar to BCS, specializers in Fabius are on a per-instruction basis and perform function inlining for tail recursive functions. It is also suggested that the specializers would perform register allocation at run-time.

Fujinami proposes a run-time specialization system for C++, targeting MIPS R4000 and Intel x86[10]. The system is designed to perform implicit optimizations; i.e., it specializes a given program with respect to its invariants, which are
determined by an automatic analysis. A specialized program runs twice as fast as the one compiled by a statically optimizing compiler. His system achieves this speedup by embedding a number of optimization algorithms into a statically generated specializer. Our approach, on the other hand, is to optimize a specialized code by using a JIT compiler, which is an independent module.

`C is a language with dynamic code generation mechanisms\cite{24}. Unlike other RTS systems, `C programmers have to explicitly specify binding-times of expressions. Similar to BCS, the implementation of `C generates programs in virtual machine languages called vCODE and iCODE. The run-time system of iCODE performs optimizations including register allocation for generated programs, similar to JIT compilers for JVMs.

Bertelsen proposes, independently of BCS, an algorithm for binding-time analysis of a JVML subset, which does not include method invocations nor objects\cite{5}. A specialization process based on the analysis is informally discussed, which is not yet implemented to the authors’ knowledge.

JSpec is an off-line, compile-time partial evaluator for Java\cite{26}. The system analyzes and specializes Java programs by applying Tempo, a partial evaluator for C, after translating the Java programs into C. This approach can be compared to ours that uses a compiler from a high-level language to a bytecode language as a front-end. Unlike current BCS implementation, JSpec supports objects whose specialization strategies are specified through specialization classes\cite{28}.

6 Conclusion

In this paper, we have proposed run-time bytecode specialization (BCS), which specializes Java virtual machine language (JVML) programs at run-time. The characteristics of this approach are summarized as follows: (1) the system directly analyzes a program and creates a specializer in an intermediate language JVML; and (2) the specializer generates programs in JVML, which makes it possible to apply optimizations after specialization by using existing JVMs with just-in-time (JIT) compilers.

The binding-time analysis algorithm is based on a type system, and also uses results of flow analysis to correctly handle stacks, local variables, and side-effects.

Thus far, we have implemented a prototype BCS system for a JVML subset and have shown that a non-trivial program specialized by our system runs approximately 3–4 times faster than the unspecialized program. The specialization cost can be amortized by 1,000 to 72,000 runs, depending on the JVMs. Those numbers are worse than the ones in the systems that are rather focusing on the specialization speed\cite{18,22}, though.

We are now extending our system to support the full JVML. Since current implementation only supports primitive types and arrays, rules that properly handle references to objects should be devised. To support objects and arrays, the system needs information whether data is modified by other methods or other threads. Such information could be obtained by either static analysis (e.g., the one studied by Choi, et al.\cite{7}) or through user declarations\cite{28}. In practice, it
is also important to support other features, such as multi-threading, and sub-
routines (i.e., jsr and ret instructions in JVML) and exceptions. Some may
consider that templates of bytecode would reduce specialization costs. As our
experiments in Section 4 showed, however, the major sources of specialization
overheads are class loading and JIT-compilation. Rather than improving the
performance of the bytecode generation process, our current plan is to generate
a specialized program directly in an intermediate language of a JIT compiler, by
using JVMs with interfaces to JIT compilers [23].

Acknowledgments

We would like to thank Reynald Affeldt, Naoya Maruyama, and Satoshi Mat-
suoka for discussing on the BCS system. We also would like to thank the anony-
mous reviewers for helpful comments.

References

   (LNCS 1694), 1999.
2. J. Auslander, M. Philipose, C. Chambers, S. J. Eggers, and B. N. Bershad. Fast,
4. A. A. Berlin and R. J. Surati. Partial evaluation for scientific computing: The
6. P. Bothner. Kawa – compiling dynamic languages to the Java VM. In USENIX,
   1998.
8. C. Consel and F. Noël. A general approach for run-time specialization and its
   application to C. In POPL’96, pp.145–170, 1996.
9. D. R. Engler. VCODE: a retargetable, extensible, very fast dynamic code genera-
10. N. Fujinami. Determination of dynamic method dispatches using run-time code
11. Y. Futamura. Partial evaluation of computation process—an approach to a
12. B. Grant, M. Philipose, M. Mock, C. Chambers, and S. J. Eggers. An evaluation
13. B. Guenter, T. B. Knoblock, and E. Ruf. Specializing shaders. In SIGGRAPH’95,
14. F. Henglein. Efficient type inference for higher-order binding-time analysis. In
15. L. Hornof and T. Jim. Certifying compilation and run-time code generation. In
A New Numerical Abstract Domain
Based on Difference-Bound Matrices

Antoine Miné
École Normale Supérieure de Paris, France,
mine@di.ens.fr,
http://www.di.ens.fr/~mine

Abstract. This paper presents a new numerical abstract domain for static analysis by abstract interpretation. This domain allows us to represent invariants of the form \((x - y \leq c)\) and \((\pm x \leq c)\), where \(x\) and \(y\) are variables values and \(c\) is an integer or real constant. Abstract elements are represented by Difference-Bound Matrices, widely used by model-checkers, but we had to design new operators to meet the needs of abstract interpretation. The result is a complete lattice of infinite height featuring widening, narrowing and common transfer functions. We focus on giving an efficient \(O(n^2)\) representation and graph-based \(O(n^3)\) algorithms—where \(n\) is the number of variables—and claim that this domain always performs more precisely than the well-known interval domain.

To illustrate the precision/cost tradeoff of this domain, we have implemented simple abstract interpreters for toy imperative and parallel languages which allowed us to prove some non-trivial algorithms correct.

1 Introduction

Abstract interpretation has proved to be a useful tool for eliminating bugs in software because it allows the design of automatic and sound analyzers for real-life programming languages. While abstract interpretation is a very general framework, we will be interested here only in discovering numerical invariants, that is to say, arithmetic relations that hold between numerical variables in a program. Such invariants are useful for tracking common errors such as division by zero and out-of-bound array access.

In this paper we propose practical algorithms to discover invariants of the form \((x - y \leq c)\) and \((\pm x \leq c)\)—where \(x\) and \(y\) are numerical program variables and \(c\) is a numeric constant. Our method works for integers, reals and even rationals.

For the sake of brevity, we will omit proofs of theorems in this paper. The complete proof for all theorems can be found in the author’s MS thesis \[12\].

Previous and Related Work. Static analysis has developed approaches to automatically find numerical invariants based on numerical abstract domains

© Springer-Verlag Berlin Heidelberg 2001
representing the form of the invariants we want to find. Famous examples are the lattice of intervals (described in, for instance, Cousot and Cousot’s ISOP’76 paper [4]) and the lattice of polyhedra (described in Cousot and Halbwachs’s POPL’78 paper [8]) which represent respectively invariants of the form \((v \in [c_1, c_2])\) and \((\alpha_1 v_1 + \cdots + \alpha_n v_n \leq c)\). Whereas the interval analysis is very efficient—linear memory and time cost—but not very precise, the polyhedron analysis is much more precise but has a huge memory cost—exponential in the number of variables.

Invariants of the form \((x - y \leq c)\) and \((\pm x \leq c)\) are widely used by the model-checking community. A special representation, called Difference-Bound Matrices (DBMs), was introduced, as well as many operators in order to model-check timed automata (see Yovine’s ES’98 paper [14] and Larsen, Larsson, Pettersson and Yi’s RTSS’97 paper [10]). Unfortunately, most operators are tied to model-checking and are of little interest for static analysis.

**Our Contribution.** This paper presents a new abstract numerical domain based on the DBM representation, together with a full set of new operators and transfer functions adapted to static analysis.

Sections 2 and 3 present a few well-known results about potential constraint sets and introduce briefly the Difference-Bound Matrices. Section 4 presents operators and transfer functions that are new—except for the intersection operator—and adapted to abstract interpretation. In Section 5, we use these operators to build lattices, which can be complete under certain conditions. Section 6 shows some practical results we obtained with an example implementation and Section 7 gives some ideas for improvement.

## 2 Difference-Bound Matrices

Let \(V = \{v_1, \ldots, v_n\}\) be a finite set a variables with value in a numerical set \(\mathbb{I}\) (which can be the set \(\mathbb{Z}\) of integers, the set \(\mathbb{Q}\) of rationals or the set \(\mathbb{R}\) of reals).

We focus, in this paper, on the representation of constraints of the form \((v_j - v_i \leq c), (v_i \leq c)\) and \((v_i \geq c)\), where \(v_i, v_j \in V\) and \(c \in \mathbb{I}\). By choosing one variable to be always equal to 0, we can represent the above constraints using only potential constraints, that is to say, constraints of the form \((v_j - v_i \leq c)\). From now, we will choose \(v_2, \ldots, v_n\) to be program variables, and \(v_1\) to be the constant 0 so that \((v_i \leq c)\) and \((v_i \geq c)\) are rewritten \((v_i - v_1 \leq c)\) and \((v_1 - v_i \leq -c)\). We assume we now work only with potential constraints over the set \(\{v_1, \ldots, v_n\}\).

**Difference-Bound Matrices.** We extend \(\mathbb{I}\) to \(\mathbb{I} = \mathbb{I} \cup \{+\infty\}\) by adding the \(+\infty\) element. The standard operations \(\leq, =, +\), min and max are extended to \(\mathbb{I}\) as usual (we will not use operations, such as \(-\) or \(*\), that may lead to indeterminate forms).

Any set \(C\) of potential constraints over \(V\) can be represented uniquely by a \(n \times n\) matrix in \(\mathbb{I}\)—provided we assume, without loss of generality, that there does not
exist two potential constraints \( (v_j - v_i \leq c) \) in \( C \) with the same left member and different right members. The matrix \( m \) associated with the potential constraint set \( C \) is called a Difference-Bound Matrix (DBM) and is defined as follows:

\[
m_{ij} \triangleq \begin{cases} 
c & \text{if } (v_j - v_i \leq c) \in C, 
+\infty & \text{elsewhere}.
\end{cases}
\]

**Potential Graphs.** A DBM \( m \) can be seen as the adjacency matrix of a directed graph \( G = (\mathcal{V}, \mathcal{A}, w) \) with edges weighted in \( \mathbb{I} \). \( \mathcal{V} \) is the set of nodes, \( \mathcal{A} \subseteq \mathcal{V}^2 \) is the set of edges and \( w: \mathcal{A} \rightarrow \mathbb{I} \) is the weight function. \( G \) is defined by:

\[
\begin{align*}
(v_i, v_j) \notin \mathcal{A} & \quad \text{if } m_{ij} = +\infty, \\
(v_i, v_j) \in \mathcal{A} \text{ and } w(v_i, v_j) = m_{ij} & \quad \text{if } m_{ij} \neq +\infty.
\end{align*}
\]

We will denote by \( \langle i_1, \ldots, i_k \rangle \) a finite set of nodes representing a path from node \( v_{i_1} \) to node \( v_{i_k} \) in \( G \). A cycle is a path such that \( i_1 = i_k \).

**\( \mathcal{V} \)-Domain and \( \mathcal{V}^0 \)-Domain.** We call the \( \mathcal{V} \)-domain of a DBM \( m \) and we denote by \( \mathcal{D}(m) \) the set of points in \( \mathbb{I}^n \) that satisfy all potential constraints:

\[
\mathcal{D}(m) \triangleq \left\{ (x_1, \ldots, x_n) \in \mathbb{I}^n \mid \forall i, j, x_j - x_i \leq m_{ij} \right\}.
\]

Now, remember that the variable \( v_1 \) has a special semantics: it is always equal to 0. Thus, it is not the \( \mathcal{V} \)-domain which is of interest, but the \( \mathcal{V}^0 \)-domain (which is a sort of intersection-projection of the \( \mathcal{V} \)-domain) denoted by \( \mathcal{D}^0(m) \) and defined by:

\[
\mathcal{D}^0(m) \triangleq \left\{ (x_2, \ldots, x_n) \in \mathbb{I}^{n-1} \mid (0, x_2, \ldots, x_n) \in \mathcal{D}(m) \right\}.
\]

We will call \( \mathcal{V} \)-domain and \( \mathcal{V}^0 \)-domain any subset of \( \mathbb{I}^n \) or \( \mathbb{I}^{n-1} \) which is respectively the \( \mathcal{V} \)-domain or the \( \mathcal{V}^0 \)-domain of some DBM. Figure 1 shows an example DBM together with its corresponding potential graph, constraint set, \( \mathcal{V} \)-domain and \( \mathcal{V}^0 \)-domain.

**\( \leq \) Order.** The \( \leq \) order on \( \mathbb{I} \) induces a point-wise order \( \leq \) on the set of DBMs:

\[
m \leq n \iff \forall i, j, m_{ij} \leq n_{ij}.
\]

This order is partial. It is also complete if \( \mathbb{I} \) has least-upper bounds, i.e., if \( \mathbb{I} \) is \( \mathbb{R} \) or \( \mathbb{Z} \), but not \( \mathbb{Q} \). We will denote by = the associated equality relation which is simply the matrix equality.

We have \( m \leq n \implies \mathcal{D}^0(m) \subseteq \mathcal{D}^0(n) \) but the converse is not true. In particular, we do not have \( \mathcal{D}^0(m) = \mathcal{D}^0(n) \implies m = n \) (see Figure 2 for a counter-example).
Fig. 1. A constraint set (a), its corresponding DBM (b) and potential graph (c), its $V$-domain (d) and $V^0$-domain (e).

Fig. 2. Three different DBMs with the same $V^0$-domain as in Figure 1. Remark that (a) and (b) are not even comparable with respect to $\preceq$.

3 Closure, Emptiness, Inclusion, and Equality Tests

We saw in Figure 2 that two different DBMs can represent the same $V^0$-domain. In this section, we show that there exists a normal form for any DBM with a non-empty $V^0$-domain and present an algorithm to find it. The existence and computability of a normal form is very important since it is, as often in abstract representations, the key to equality testing used in fixpoint computation. In the case of DBMs, it will also allow us to carry an analysis of the precision of the operators defined in the next section.
Emptiness Testing. We have the following graph-oriented theorem:

**Theorem 1.**
A DBM has an empty $V^0$-domain if and only if there exists, in its associated potential graph, a cycle with a strictly negative total weight.

Checking for cycles with a strictly negative weight is done using the well-known *Bellman-Ford algorithm* which runs in $O(n^3)$. This algorithm can be found in Cormen, Leiserson and Rivest’s classical algorithmics textbook [2, §25.3].

Closure and Normal Form. Let $m$ be a DBM with a non-empty $V^0$-domain and $G$ its associated potential graph. Since $G$ has no cycle with a strictly negative weight, we can compute its *shortest path closure* $G^*$, the adjacency matrix of which will be denoted by $m^*$ and defined by:

$$
\begin{cases}
  m^*_{ii} \triangleq 0, \\
  m^*_{ij} \triangleq \min_{1 \leq k \leq N} \sum_{i=1}^{N-1} m_{ik}^{i+1} & \text{if } i \neq j.
\end{cases}
$$

The idea of closure relies on the fact that, if $\langle i = i_1, i_2, \ldots, i_N = j \rangle$ is a path from $v_i$ to $v_j$, then the constraint $v_j - v_i \leq \sum_{k=1}^{N-1} m_{ik}^{i+1}$ can be derived from $m$ by adding the potential constraints $v_{ik}^{i+1} - v_{ik} \leq m_{ik}^{i+1}, 1 \leq k \leq N - 1$. This is an *implicit* potential constraint which does not appear directly in the DBM $m$. When computing the closure, we replace each potential constraint $v_j - v_i \leq m_{ij}, i \neq j$ in $m$ by the tightest implicit constraint we can find, and each diagonal element by 0 (which is indeed the smallest value $v_i - v_i$ can reach).

In Figure 2 for instance, (c) is the closure of both the (a) and (b) DBMs.

**Theorem 2.**

1. $m^* = \inf \{ n \mid D^0(n) = D^0(m) \}$.
2. $D^0(m)$ saturates $m^*$, that is to say:

   \[ \forall i, j, \text{ such that } m^*_{ij} < +\infty, \exists (x_1 = 0, x_2, \ldots, x_n) \in D(m), x_j - x_i = m^*_{ij}. \]

   \[ \square \]

Theorem 2.1 states that $m^*$ is the smallest DBM—with respect to $\preceq$—that represents a given $V^0$-domain, and thus the closed form is a normal form. Theorem 2.2 is a crucial property to prove accuracy of some operators defined in the next section.

Any shortest-path graph algorithm can be used to compute the closure of a DBM. We suggest the straightforward *Floyd-Warshall*, which is described in Cormen, Leiserson and Rivest’s textbook [2] §26.2, and has a $O(n^3)$ time cost.
Equality and Inclusion Testing. The case where \( m \) or \( n \) or both have an empty \( \mathcal{V}_0 \)-domain is easy; in all other cases we use the following theorem—which is a consequence of Theorem 2.1:

**Theorem 3.**

1. If \( m \) and \( n \) have non-empty \( \mathcal{V}_0 \)-domain, \( \mathcal{D}_0(m) = \mathcal{D}_0(n) \iff m^* = n^* \).
2. If \( m \) and \( n \) have non-empty \( \mathcal{V}_0 \)-domain, \( \mathcal{D}_0(m) \subseteq \mathcal{D}_0(n) \iff m^* \preceq n^* \).

Besides emptiness test and closure, we may need, in order to test equality or inclusion, to compare matrices with respect to the point-wise ordering \( \leq \). This can be done with a \( \mathcal{O}(n^2) \) time cost.

**Projection.** We define the **projection** \( \pi_{v_k}(m) \) of a DBM \( m \) with respect to a variable \( v_k \) to be the interval containing all possible values of \( v \in \mathbb{I} \) such that there exists a point \((x_2, \ldots, x_n)\) in the \( \mathcal{V}_0 \)-domain of \( m \) with \( x_k = v \):

\[
\pi_{v_k}(m) \triangleq \{ x \in \mathbb{I} \mid \exists (x_2, \ldots, x_n) \in \mathcal{D}_0(m) \text{ such that } x = x_k \}.
\]

The following theorem, which is a consequence of the saturation property of the closure, gives an algorithmic way to compute the projection:

**Theorem 4.**

If \( m \) has a non-empty \( \mathcal{V}_0 \)-domain, then \( \pi_{v_k}(m) = [-m^*_{k1}, m^*_{1k}] \)
(interval bounds are included only if finite).

### 4 Operators and Transfer Functions

In this section, we define some operators and transfer functions to be used in abstract semantics. Except for the intersection operator, they are new. The operators are basically point-wise extensions of the standard operators defined over the domain of intervals \([4]\).

Most algorithms presented here are either constant time, or point-wise, i.e., quadratic time.

**Intersection.** Let us define the point-wise intersection DBM \( m \land n \) by:

\[
(m \land n)_{ij} \triangleq \min(m_{ij}, n_{ij})
\]

We have the following theorem:

**Theorem 5.**

\[
\mathcal{D}_0(m \land n) = \mathcal{D}_0(m) \cap \mathcal{D}_0(n).
\]

stating that the intersection is always exact. However, the resulting DBM is seldom closed, even if the arguments are closed.
Least Upper Bound. The set of $\mathcal{V}^0$-domains is not stable by union so we introduce here a union operator which over-approximate its result. We define the point-wise least upper bound DBM $m \lor n$ by:

$$(m \lor n)_{ij} \triangleq \max(m_{ij}, n_{ij}).$$

$m \lor n$ is indeed the least upper bound with respect to the $\subseteq$ order. The following theorem tells us about the effect of this operator on $\mathcal{V}^0$-domains:

Theorem 6.

1. $D^0(m \lor n) \supseteq D^0(m) \cup D^0(n)$.
2. If $m$ and $n$ have non-empty $\mathcal{V}^0$-domains, then

$$(m^*) \lor (n^*) = \inf_{\subseteq} \{ o \mid D^0(o) \supseteq D^0(m) \cup D^0(n) \}$$

and, as a consequence, $D^0((m^*) \lor (n^*))$ is the smallest $\mathcal{V}^0$-domain (with respect to the $\subseteq$ ordering) which contains $D^0(m) \cup D^0(n)$.
3. If $m$ and $n$ are closed, then so is $m \lor n$.

Widening. When computing the semantics of a program, one often encounters loops leading to fixpoint computation involving infinite iteration sequences. In order to compute in finite time an upper approximation of a fixpoint, widening operators were introduced in P. Cousot’s thesis [3, §4.1.2.0.4]. Widening is a sort of union for which every increasing chain is stationary after a finite number of iterations. We define the point-wise widening operator $\triangleleft$ by:

$$(m \triangledown n)_{ij} \triangleq \begin{cases} m_{ij} & \text{if } n_{ij} \leq m_{ij}, \\ +\infty & \text{elsewhere}. \end{cases}$$

The following properties prove that $\triangledown$ is indeed a widening:

Theorem 7.

1. $D^0(m \triangledown n) \supseteq D^0(m) \cup D^0(n)$.
2. Finite chain property:

$\forall m \text{ and } \forall (n_i)_{i \in \mathbb{N}}$, the chain defined by:

$1$ $\mathcal{V}^0$-domains are always convex, but the union of two $\mathcal{V}^0$-domains may not be convex.
\[
\begin{align*}
  x_0 & \triangleq m, \\
  x_{i+1} & \triangleq x_i \triangledown n_i,
\end{align*}
\]

is increasing for \( \sqsubseteq \) and ultimately stationary. The limit \( l \) is such that \( l \triangleright m \) and \( \forall i, l \triangleright n_i \).

The widening operator has some intriguing interactions with closure. Like the least upper bound, the widening operator gives more precise results if its right argument is closed, so it is rewarding to change \( x_{i+1} = x_i \triangledown n_i \) into \( x_{i+1} = x_i \triangledown (n_i^*) \). This is not the case for the first argument: we can have sometimes \( D^0(m \triangledown n) \subsetneq D^0((m^*) \triangledown n) \). Worse, if we try to force the closure of the first argument by changing \( x_{i+1} = x_i \triangledown n_i \) into \( x_{i+1} = (x_i \triangledown n_i)^* \), the finite chain property (Theorem 7.2) is no longer satisfied, as illustrated in Figure 3.

**Fig. 3.** Example of an infinite strictly increasing chain defined by \( x_0 = m^* \), \( x_{i+1} = (x_i \triangledown n_i)^* \).

Originally [4], Cousot and Cousot defined widening over intervals \( \triangledown \) by:

\[
[a, b] \triangledown [c, d] \triangleq [e, f],
\]

where:

\[
e \triangleq \begin{cases} a & \text{if } a \leq c, \\ -\infty & \text{elsewhere,} \end{cases} \quad f \triangleq \begin{cases} b & \text{if } b \geq d, \\ +\infty & \text{elsewhere} \end{cases}.
\]

The following theorem proves that the sequence computed by our widening is always more precise than with the standard widening over intervals:
Theorem 8.
If we have the following iterating sequence:
\[
\begin{align*}
  x_0 &\triangleq m^*, \\
  x_{k+1} &\triangleq x_k \triangledown (n_k^*),
\end{align*}
\]
\[
\begin{align*}
  [y_0, z_0] &\triangleq \pi_{[v_i]}(m), \\
  [y_{k+1}, z_{k+1}] &\triangleq [y_k, z_k] \triangledown \pi_{[v_i]}(n_k),
\end{align*}
\]
then the sequence \((x_k)_{k \in \mathbb{N}}\) is more precise than the sequence \(([y_k, z_k])_{k \in \mathbb{N}}\) in the following sense:
\[
\forall k, \pi_{[v_i]}(x_k) \subseteq [y_k, z_k].
\]
\]

Remark that the technique, described in Cousot and Cousot’s PLILP’92 paper [7], for improving the precision of the standard widening over intervals \(\triangledown\) can also be applied to our widening \(\triangledown\). It allows, for instance, deriving a widening that always gives better results than a simple sign analysis (which is not the case of \(\triangledown\) or \(\triangledown\)). The resulting widening over DBMs will remain more precise than the resulting widening over intervals.

Narrowing. Narrowing operators were introduced in P. Cousot’s thesis [3, §4.1.2.0.11] in order to restore, in a finite time, some information that may have been lost by widening applications. We define here a point-wise narrowing operator \(\triangle\) by:
\[
(m \triangle n)_{ij} \triangleq \begin{cases} 
  n_{ij} & \text{if } m_{ij} = +\infty, \\
  m_{ij} & \text{elsewhere}.
\end{cases}
\]

The following properties prove that \(\triangle\) is indeed a narrowing:

Theorem 9.
1. If \(D^0(n) \subseteq D^0(m)\), then \(D^0(n) \subseteq D^0(m \triangle n) \subseteq D^0(m)\).
2. Finite decreasing chain property:
   \(\forall m\) and for any chain \((n_i)_{i \in \mathbb{N}}\) decreasing for \(\subseteq\), the chain defined by:
   \[
   \begin{align*}
   x_0 &\triangleq m, \\
   x_{i+1} &\triangleq x_i \triangledown n_i,
   \end{align*}
   \]
   is decreasing and ultimately stationary.

Given a sequence \((n_k)_{k \in \mathbb{N}}\) such that the chain \((D^0(n_k))_{k \in \mathbb{N}}\) is decreasing for the \(\subseteq\) partial order (but not \((n_k)_{k \in \mathbb{N}}\) for the \(\triangle\) partial order), one way to ensure the best accuracy as well as the finiteness of the chain \((x_k)_{k \in \mathbb{N}}\) is to force the closure of the right argument by changing \(x_{i+1} = x_i \triangle n_i\) into \(x_{i+1} = x_i \triangle (n_i^*)\). Unlike widening, forcing all elements in the chain to be closed with \(x_{i+1} = (x_i \triangle n_i)^*\) poses no problem.
Forget. Given a DBM $m$ and a variable $v_k$, the forget operator $m\backslash v_k$ computes a DBM where all informations about $v_k$ are lost. It is the opposite of the projection operator $\pi_{v_k}$. We define this operator by:

$$(m\backslash v_k)_{ij} \triangleq \begin{cases} 
\min(m_{ij}, m_{ik} + m_{kj}) & \text{if } i \neq k \text{ and } j \neq k, \\
0 & \text{if } i = j = k, \\
+\infty & \text{elsewhere}.
\end{cases}$$

The $v^0$-domain of $m\backslash v_k$ is obtained by projecting $D^0(m)$ on the subspace orthogonal to $\mathbb{P}\overrightarrow{v_k}$, and then extruding the result in the direction of $\overrightarrow{v_k}$:

**Theorem 10.**

$$D^0(m\backslash v_k) = \{(x_2, \ldots, x_n) \in \mathbb{P}^{n-1} | \exists x \in \mathbb{P}, (x_2, \ldots, x_{k-1}, x, x_{k+1}, \ldots, x_n) \in D^0(m)\}. \square$$

Guard. Given an arithmetic equality or inequality $g$ over $\{v_2, \ldots, v_n\}$—which we call a guard—and a DBM $m$, the guard transfer function tries to find a new DBM $m(g)$ the $v^0$-domain of which is $\{s \in D^0(m) | s \text{ satisfies } g\}$. Since this is, in general, impossible, we will only try to have:

**Theorem 11.**

$$D^0(m(g)) \supseteq \{s \in D^0(m) | s \text{ satisfies } g\}. \square$$

Here is an example definition:

**Definition 12.**

1. If $g = (v_{j_0} - v_{i_0} \leq c)$ with $i_0 \neq j_0$, then:

$$(m(v_{j_0} - v_{i_0} \leq c))_{ij} \triangleq \begin{cases} 
\min(m_{ij}, c) & \text{if } i = i_0 \text{ and } j = j_0, \\
m_{ij} & \text{elsewhere}.
\end{cases}$$

The cases $g = (v_{j_0} \leq c)$ and $g = (-v_{i_0} \leq c)$ are settled by choosing respectively $i_0 = 1$ and $j_0 = 1$.

2. If $g = (v_{j_0} - v_{i_0} = c)$ with $i_0 \neq j_0$, then:

$$m(v_{j_0} - v_{i_0} = c) \triangleq (m(v_{j_0} - v_{i_0} \leq c))(v_{i_0} - v_{j_0} \leq c) .$$

The case $g = (v_{j_0} = c)$ is a special case where $i_0 = 1$.

3. In all other cases, we simply choose:

$$m(g) \triangleq m.$$ 

In all but the last—general—cases, the guard transfer function is exact.
Assignment. An assignment \( v_k \leftarrow e(v_2, \ldots, v_n) \) is defined by a variable \( v_k \) and an arithmetic expression \( e \) over \( \{v_2, \ldots, v_n\} \).

Given a DBM \( m \) representing all possible values that can take the variables set \( \{v_2, \ldots, v_n\} \) at a program point, we look for a DBM, denoted by \( m(v_k\leftarrow e) \), representing the possible values of the same variables set after the assignment \( v_k \leftarrow e \). This is not possible in the general case, so the assignment transfer function will only try to find an upper approximation of this set:

Theorem 13.

\[
D^0(m(v_k\leftarrow e)) \supseteq \{(x_2, \ldots, x_{k-1}, e(x_2, \ldots, x_n), x_{k+1}, \ldots, x_n) \mid (x_2, \ldots, x_n) \in D^0(m)\}. \quad \square
\]

For instance, we can use the following definition for \( m(v_{i_0}\leftarrow e) \):

Definition 14.

1. If \( e = v_{i_0} + c \), then:

\[
(m(v_{i_0}\leftarrow v_{i_0} + c))_{ij} \triangleq \begin{cases} m_{ij} - c & \text{if } i = i_0, j \neq j_0, \\ m_{ij} + c & \text{if } i \neq i_0, j = j_0, \\ m_{ij} & \text{elsewhere}. \end{cases}
\]

2. If \( e = v_{j_0} + c \) with \( i_0 \neq j_0 \), then we use the forget operator and the guard transfer function:

\[
m(v_{i_0}\leftarrow v_{j_0} + c) \triangleq ((m\setminus v_{i_0})(v_{i_0} - v_{j_0} \leq c))(v_{j_0} - v_{i_0} \leq -c).
\]

The case \( e = c \) is a special case where we choose \( j_0 = 1 \).

3. In all other cases, we use a standard interval arithmetic to find an interval \([e^-, e^+], e^+, e^- \in \mathbb{F} \) such that

\[
[-e^-, e^+] \supseteq e(\pi_{v_2}(m), \ldots, \pi_{v_n}(m))
\]

and then we define:

\[
(m(v_{i_0}\leftarrow e))_{ij} \triangleq \begin{cases} e^+ & \text{if } i = 1 \text{ and } j = i_0, \\ e^- & \text{if } j = 1 \text{ and } i = i_0, \\ (m\setminus v_{i_0})_{ij} & \text{elsewhere}. \end{cases}
\]

In all but the last—general—cases, the assignment transfer function is exact.
Comparison with the Abstract Domain of Intervals. Most of the time, the precision of numerical abstract domains can only be compared experimentally on example programs (see Section 6 for such an example). However, we claim that the DBM domain always performs better than the domain of intervals.

To legitimize this assertion, we compare informally the effect of all abstract operations in the DBM and in the interval domains. Thanks to Theorems 5 and 6.2, and Definitions 12 and 14, the intersection and union abstract operators and the guard and assignment transfer functions are more precise than their interval counterpart. Thanks to Theorem 8, approximate fixpoint computation with our widening \( \varpi \) is always more accurate than with the standard widening over intervals \( \varpi \) and one could prove easily that each iteration with our narrowing is more precise than with the standard narrowing over intervals. This means that any abstract semantics based on the operators and transfer functions we defined is always more precise than the corresponding interval-based abstract semantics.

## 5 Lattice Structures

In this section, we design two lattice structures: one on the set of DBMs and one on the set of closed DBMs. The first one is useful to analyze fixpoint transfer between abstract and concrete semantics and the second one allows us to design a meaning function—or even a Galois Connection—linking the set of abstract \( \mathcal{V}_0 \)-domains to the concrete lattice \( \mathcal{P}(\{v_2, \ldots, v_n\} \mapsto \mathbb{I}) \), following the abstract interpretation framework described in Cousot and Cousot’s POPL’79 paper.

**DBM Lattice.** The set \( \mathcal{M} \) of DBMs, together with the order relation \( \sqsubseteq \) and the point-wise least upper bound \( \lor \) and greatest lower bound \( \land \), is almost a lattice. It only needs a least element \( \perp \), so we extend \( \sqsubseteq \), \( \lor \) and \( \land \) to \( \mathcal{M}_\perp = \mathcal{M} \cup \{\perp\} \) in an obvious way to get \( \sqsubseteq \), \( \lor \) and \( \land \). The greatest element \( \top \) is the DBM with all its coefficients equal to \( +\infty \).

**Theorem 15.**

1. \((\mathcal{M}_\perp, \sqsubseteq, \sqcap, \sqcup, \perp, \top)\) is a lattice.
2. This lattice is complete if \((\mathbb{I}, \leq)\) is complete (\( \mathbb{Z} \) or \( \mathbb{R} \), but not \( \mathbb{Q} \)).

\(\Box\)

There are, however, two problems with this lattice. First, we cannot easily assimilate this lattice to a sub-lattice of \( \mathcal{P}(\{v_2, \ldots, v_n\} \mapsto \mathbb{I}) \) as two different DBMs can have the same \( \mathcal{V}_0 \)-domain. Then, the least upper bound operator \( \sqcup \) is not the most precise upper approximation of the union of two \( \mathcal{V}_0 \)-domains because we do not force the arguments to be closed.
Closed DBM Lattice. To overcome these difficulties, we build another lattice based on closed DBMs. First, consider the set \( \mathcal{M}^* \) of closed DBMs with a least element \( \bot^* \) added. Now, we define a greatest element \( \top^* \), a partial order relation \( \sqsubseteq^* \), a least upper bound \( \sqcup^* \) and a greatest lower bound \( \sqcap^* \) in \( \mathcal{M}^*_\bot \) by:

\[
\begin{align*}
\top^*_{ij} & \equiv \begin{cases} 
0 & \text{if } i = j, \\
+\infty & \text{elsewhere}.
\end{cases} \\
\bot^* & \equiv \{ \text{either } m = \bot^*, \\
& \text{or } m \neq \bot^*, n \neq \bot^* \text{ and } m \sqsubseteq n \}.
\end{align*}
\]

\[
\begin{align*}
\bot^* & \equiv \{ \text{either } m = \bot^*, \\
& \text{or } m \neq \bot^*, n \neq \bot^* \text{ and } m \sqsubseteq n \}.
\end{align*}
\]

\[
\begin{align*}
m \sqcup^* n & \equiv \begin{cases} 
 m & \text{if } n = \bot^*, \\
 n & \text{if } m = \bot^*, \\
 m \lor n & \text{elsewhere}.
\end{cases} \\
m \sqcap^* n & \equiv \begin{cases} 
 \bot^* & \text{if } m = \bot^* \text{ or } n = \bot^* \text{ or } D^0(m \land n) = \emptyset, \\
 (m \land n)^* & \text{elsewhere}.
\end{cases}
\end{align*}
\]

Thanks to Theorem 2.1, every non-empty \( \gamma^0 \)-domain has a unique representation in \( \mathcal{M}^* \); \( \bot^* \) is the representation for the empty set. We build a meaning function \( \gamma \) which is an extension of \( D^0(\cdot) \) to \( \mathcal{M}^*_\bot \):

\[
\gamma(m) \equiv \begin{cases} 
\emptyset & \text{if } m = \bot^*, \\
D^0(m) & \text{elsewhere}.
\end{cases}
\]

Theorem 16.

1. \( (\mathcal{M}^*_\bot, \sqsubseteq^*, \sqcap^*, \sqcup^*, \bot^*, \top^*) \) is a lattice and \( \gamma \) is one-to-one.

2. If \( (\mathbb{I}, \leq) \) is complete, this lattice is complete and \( \gamma \) is meet-preserving:

\[
\gamma(\sqcap^* X) = \bigcap \{ \gamma(x) \mid x \in X \}. 
\]

We can—according to Cousot and Cousot [6, Prop. 7]—build a canonical Galois Insertion:

\[
\mathcal{P}(\{v_2, \ldots, v_n\} \mapsto \mathbb{I}) \overset{\gamma}{\longrightarrow} \mathcal{M}^*_\bot
\]

where the abstraction function \( \alpha \) is defined by:

\[
\alpha(D) = \sqcap^* \{ m \in \mathcal{M}^*_\bot \mid D \subseteq \gamma(m) \}.
\]

The \( \mathcal{M}^*_\bot \) lattice features a nice meaning function and a precise union approximation; thus, it is tempting to force all our operators and transfer functions to live in \( \mathcal{M}^*_\bot \) by forcing closure on their result. However, we saw this does not work for widening, so fixpoint computation must be performed in the \( \mathcal{M}^*_\bot \) lattice.

6 Results

The algorithms on DBMs presented here have been implemented in OCaml and used to perform forward analysis on toy—yet Turing-equivalent—imperative and parallel languages with only numerical variables and no procedure.
We present here neither the concrete and abstract semantics, nor the actual forward analysis algorithm used for our analyzers. They follow exactly the abstract interpretation scheme described in Cousot and Cousot’s POPL’79 paper \cite{5} and Bourdoncle’s FMPA’93 paper \cite{1} and are detailed in the author’s MS thesis \cite{12}. Theorems 1, 3, 5, 6, 11 and 13 prove that all the operators and transfer functions we defined are indeed abstractions on the domain of DBMs of the usual operators and transfer functions on the concrete domain $P(\{v_2, \ldots, v_n\} \mapsto \bot)$, which, as shown by Cousot and Cousot \cite{5}, is sufficient to prove soundness for analyses.

**Imperative Programs.** Our toy forward analyzer for imperative language follows almost exactly the analyzer described in Cousot and Halbwachs’s POPL’78 paper \cite{8}, except that the abstract domain of polyhedra has been replaced by our DBM-based domain. We tested our analyzer on the well-known Bubble Sort and Heap Sort algorithms and managed to prove automatically that they do not produce out-of-bound error while accessing array elements. Although we did not find as many invariants as Cousot and Halbwachs for these two examples, it was sufficient to prove the correctness. We do not detail these common examples here for the sake of brevity.

**Parallel Programs.** Our toy analyzer for parallel language allows analyzing a fixed set of processes running concurrently and communicating through global variables. We use the well-known nondeterministic interleaving method in order to analyze all possible control flows. In this context, we managed to prove automatically that the Bakery algorithm, introduced in 1974 by Lamport \cite{9}, for synchronizing two parallel processes never lets the two processes be at the same time in their critical sections. We now detail this example.

**The Bakery Algorithm.** After the initialization of two global shared variables $y_1$ and $y_2$, two processes $p_1$ and $p_2$ are spawned. They synchronize through the variables $y_1$ and $y_2$, representing the priority of $p_1$ and $p_2$, so that only one process at a time can enter its critical section (Figure 4).

Our analyzer for parallel processes is fed with the initialization code ($y_1 = 0; y_2 = 0$) and the control flow graphs for $p_1$ and $p_2$ (Figure 5). Each control graph is a set of control point nodes and some edges labeled with either an action performed when the edge is taken (the assignment $y_1 \leftarrow y_2 + 1$, for example) or a guard imposing a condition for taking the edge (the test $y_1 \neq 0$, for example).

The analyzer then computes the nondeterministic interleaving of $p_1$ and $p_2$ which is the product control flow graph. Then, it computes iteratively the abstract invariants holding at each product control point. It outputs the invariants shown in Figure 6.

The state $(2, c)$ is never reached, which means that $p_1$ and $p_2$ cannot be at the same time in their critical section. This proves the correctness of the Bakery algorithm. Remark that our analyzer also discovered some non-obvious invariants, such as $y_1 = y_2 + 1$ holding in the $(1, c)$ state.
\[ y_1 = 0; \ y_2 = 0; \]

\[(p1)\]
\[
\text{while true do } \\
y_1 = y_2 + 1; \\
\text{while } y_2 \neq 0 \text{ and } y_1 > y_2 \text{ do done; } \\
--- \text{critical section} --- \\
y_1 = 0; \\
\text{done}
\]

\[(p2)\]
\[
\text{while true do } \\
y_2 = y_1 + 1; \\
\text{while } y_1 \neq 0 \text{ and } y_2 \geq y_1 \text{ do done; } \\
--- \text{critical section} --- \\
y_2 = 0; \\
\text{done}
\]

\textbf{Fig. 4.} Pseudo-code for the Bakery algorithm.

\begin{figure*}[h]
\centering
\includegraphics[width=\textwidth]{fig5}
\caption{Control flow graphs of processes \textit{p1} and \textit{p2} in the Bakery algorithm.}
\end{figure*}
7 Extensions and Future Work

Precision Improvement. In our analysis, we only find a coarse set of the invariants held in a program since finding all invariants of the form \((x - y \leq c)\) and \((\pm x \leq c)\) for all programs is non-computable. Possible losses of precision have three causes: non-exact union, widening in loops and non-exact assignment and guard transfer functions.

We made crude approximations in the last—general—case of Definitions 12 and 14 and there is room for improving assignment and guard transfer functions, even though exactness is impossible. When the DBM lattices are complete, there exists most precise transfer functions such that Theorems 11 and 13 hold, however these functions may be difficult to compute.

Finite Union of \(V^0\)-Domains. One can imagine to represent finite unions of \(V^0\)-domains, using a finite set of DBMs instead of a single one as abstract state. This allows an exact union operator but it may lead to memory and time cost explosion as abstract states contain more and more DBMs, so one may need from time to time to replace a set of DBMs by their union approximation.

The model-checker community has also developed specific structures to represent finite unions of \(V\)-domains, that are less costly than sets. Clock-Difference Diagrams (introduced in 1999 by Larsen, Weise, Yi and Pearson [11]) and Difference Decision Diagrams (introduced in Møller, Lichtenberg, Andersen and Hulgaard’s CSL’99 paper [13]) are tree-based structures made compact thanks to the sharing of isomorphic sub-trees; however existence of normal forms for such structures is only a conjecture at the time of writing and only local or...
path reduction algorithms exist. One can imagine adapting such structures to abstract interpretation the way we adapted DBM in this paper.

**Space and Time Cost Improvement.** Space is often a big concern in abstract interpretation. The DBM representation we proposed in this paper has a fixed $O(n^2)$ memory cost—where $n$ is the number of variables in the program. In the actual implementation, we decided to use the graph representation—or hollow matrix—which stores only edges with a finite weight and observed a great space gain as most DBMs we use have many $+\infty$. Most algorithms are also faster on hollow matrices and we chose to use the more complex, but more efficient, *Johnson* shortest-path closure algorithm—described in Cormen, Leiserson and Rivest’s textbook [2, §26.3]—instead of the *Floyd-Warshall* algorithm.

Larsen, Larsson, Pettersson and Yi’s RTSS’97 paper [10] presents a minimal form algorithm which finds a DBM with the fewest finite edges representing a given $V^0$-domain. This minimal form could be useful for memory-efficient storing, but cannot be used for direct computation with algorithms requiring closed DBMs.

**Representation Improvement.** The invariants we manipulate are, in term of precision and complexity, between interval and polyhedron analysis. It is interesting to look for domains allowing the representation of more forms of invariants than DBMs in order to increase the granularity of numerical domains. We are currently working on an improvement of DBMs that allows us to represent, with a small time and space complexity overhead, invariants of the form $(\pm x \pm y \leq c)$.

8 Conclusion

We presented in this paper a new numerical abstract domain inspired from the well-known domain of intervals and the Difference-Bound Matrices. This domain allows us to manipulate invariants of the form $(x - y \leq c)$, $(x \leq c)$ and $(x \geq c)$ with a $O(n^2)$ worst case memory cost per abstract state and $O(n^3)$ worst case time cost per abstract operation (where $n$ is the number of variables in the program).

Our approach made it possible for us to prove the correctness of some non-trivial algorithms beyond the scope of interval analysis, for a much smaller cost than polyhedron analysis. We also proved that this analysis always gives better results than interval analysis, for a slightly greater cost.

**Acknowledgments.** I am grateful to J. Feret, C. Hymans, D. Monniaux, P. Cousot, O. Danvy and the anonymous referees for their helpful comments and suggestions.
References

Partial Evaluation for Class-Based Object-Oriented Languages

Ulrik P. Schultz

DAIMI, University of Aarhus,
Aabogade 34, DK-8200 Aarhus N, Denmark
ups@daimi.au.dk

Abstract. Object-oriented programming facilitates the development of generic software, but at a significant cost in terms of performance. We apply partial evaluation to object-oriented programs, to automatically map generic software into specific implementations. In this paper we give a concise, formal description of a simple partial evaluator for a minimal object-oriented language, and give directions for extending this partial evaluator to handle realistic programs.

1 Introduction

The object-oriented style of programming naturally leads to the development of generic program components. Encapsulation of data and code into objects enhances code resilience to program modifications and increases the opportunities for direct code reuse. Message passing between objects lets program components communicate without relying on a specific implementation; this decoupling enables dynamic modification of the program structure in order to react to changing conditions. Genericity implemented using these language features is however achieved at the expense of efficiency. Encapsulation isolates individual program parts and increases the cost of data access. Message passing is implemented using virtual dispatching, which obscures control flow, thus blocking traditional optimizations at both the hardware and software level.

Partial evaluation is an automated technique for mapping generic programs into specific implementations dedicated to a specific purpose. Partial evaluation has been investigated extensively for functional, logical and imperative languages, and has recently been investigated for object-oriented languages by Schultz et al., in the context of a prototype partial evaluator for Java. However, no precise specification of partial evaluation for object-oriented languages has thus far been given.

In this paper, we give a concise description of the effect of partial evaluation on an object-oriented program, and formalize how an object-oriented program can be specialized using an off-line partial evaluator. The formalization is done

* Based on work done in the Compose Group at IRISA/INRIA, Rennes, France; supported in part by Bull.
on a minimal object-oriented language without side-effects, and the partial evaluator that we define has monovariant binding times and no partially static data. Nevertheless, we argue that these partial evaluation principles can be extended to specialize realistic programs written in Java. Indeed, these principles form the basis of a complete partial evaluator for Java, briefly described in Section 3 and described in detail elsewhere [24,25]. We consider class-based object-oriented languages; partial evaluation for object-based object-oriented languages is future work.

Overview: First, Section 2 gives a concise description of the effect of partial evaluation on an object-oriented program. Then, Section 3 defines a small object-oriented language based on Java. The four following sections define a partial evaluator for this language: Section 4 defines a two-level syntax, Section 5 gives well-annotatedness rules, Section 6 gives specialization rules, and Section 7 gives a constraint system for deriving well-annotated programs. Afterwards, Section 8 provides examples of how this partial evaluator can specialize small object-oriented programs, and Section 9 summarizes the features needed to scale up the partial evaluator to specialize realistic Java programs. Last, Section 10 investigates related work, and Section 11 concludes and discusses future work.

Terminology: In object-oriented programming, the word “specialize” usually means “to subclass,” and the word “static” usually indicates a class method (i.e., a method that does not have a self parameter). We here use the word specialize in a different sense, to mean the optimization of a program or a program part based on knowledge about the evaluation context. Also, we always use the word static to indicate known information.

2 Specializing Object-Oriented Programs

In this section, we first describe the basic principles for specializing object-oriented programs, and then give an example.

2.1 Basic Principles

We first explain how to specialize a program by specializing its methods, then explain how to generate a specialized method, and last explain how to reintegrate specialized methods into the program.

Globally, the execution of an object-oriented program can be seen as a sequence of interactions between the objects that constitute the program. Parts of this interaction may become fixed when particular program input parameters are fixed. Given fixed program input parameters, partial evaluation can specialize the program by simplifying the object interaction as much as possible. The static (known) interactions can be evaluated, leaving behind only the dynamic (unknown) interactions.
Objects interact by using virtual dispatches to invoke methods. We can specialize the interaction that takes place between a collection of objects by specializing their methods for any static arguments. Each specialized method is added to the object where the corresponding generic method is defined. Using this approach, the specialized object interaction is expressed in terms of the specialized methods: a specialized method interacts with some object by calling specialized methods on this object.

A method is specialized to a set of static values by propagating these values throughout the body, and using them to reduce field lookups, method invocations, and non-object computations. A lookup of a static value stored in a field of a static object yields a value that can be used to specialize other parts of the program. A virtual dispatch is akin to a conditional that tests the type of the receiver object and subsequently calls the appropriate receiver method. When the receiver object is static, the virtual dispatch can be eliminated, and the body of the method unfolded into the caller. When the receiver object is dynamic but is passed static arguments, the virtual dispatch can be specialized speculatively; each potential receiver method is specialized for the static arguments, and a virtual dispatch to the specialized methods is residualized. Object-oriented languages often include features from functional or imperative languages; such features can be specialized according to the known partial evaluation principles for these languages.

The result of specializing a program is a collection of specialized methods to be introduced into the classes of the program. However, introducing the specialized methods directly into the classes of the program is problematic: encapsulation invariants may be broken by specialized methods where safety checks have been specialized away, and this mix of generic and specialized code obfuscates the appearance of the program and complicates maintenance. A representation of the specialized program is needed that preserves encapsulation and modularity.

We observe that the dependencies between the specialized methods follow the control flow of the program, which cuts across the class structure of the program. This observation brings aspect-oriented programming to mind: aspect-oriented programming allows logical units that cut across the program structure to be separated from other parts of the program and encapsulated into an aspect. The methods generated by a given specialization of an object-oriented program can be encapsulated into a separate aspect, and only woven into the program during compilation. Access modifiers can be used to ensure that specialized methods only can be called from specialized methods encapsulated in the same aspect, and hence always are called from a safe context. Furthermore, the specialized code is cleanly separated from the generic code, and can be plugged and unplugged by selecting whether to include the aspect in the program.

In this paper, we represent specialized programs using an aspect syntax based on the AspectJ language. In this syntax, a specialized program is a named aspect which holds a number of introduction blocks. Each introduction block lists a set of methods to introduce into the class named by the block header. Note that to permit a standard compiler to be used, a weaver will usually produce a
class Power {
    int exp; Binary op; int neutral;
    Power( int exp, Binary op, int neutral ) {
        super();
        this.exp = exp; this.op = op;
        this.neutral = neutral;
    }
    int raise( int base ) {
        return loop(base,this.exp);
    }
    int loop( int base, int x ) {
        return x==0
            ? this.neutral
            : this.op.e( base,
                        this.loop( base, x-1 ) );
    }
}

class Binary {  
    int e( int x, int y ) {
        return this.e(x,y);
    }
}

class Add extends Binary {  
    int e( int x, int y ) {
        return x+y;
    }
}

class Mul extends Binary {  
    int e( int x, int y ) {
        return x*y;
    }
}

Fig. 1. A power function and binary operators.

standard, object-oriented program. Also, if whole-program specialization is used, the set of specialized methods will be self-contained, and the aspect syntax would thus be redundant.

2.2 Example: Power

As an example of how partial evaluation for object-oriented languages specializes a program, we use the collection of Java classes shown in Figure 1. These classes implement an object-oriented version of the power function, parameterized by the exponent, the binary operator to apply, and the neutral value. The power function is computed by the method raise of the class Power; this method uses the recursive method loop to repeatedly apply a binary operator to a value. The binary operator functionality is delegated to a Binary object, following the Strategy design pattern [10]. The class Binary is the common superclass of the two binary operators Add and Mul.

We can specialize the method raise of the class Power in a number of ways; the results are illustrated in Figure 2. First, assume that the exponent field is known; propagating the value stored in the exponent field throughout the program allows the recursion of the method raise to be unfolded. The result is shown in the aspect Exp_Known. Next, we can specialize the program according to a different context where the operator and neutral values also are known; the virtual dispatch to the binary operator can be resolved and unfolded, and the neutral value

1 Rather than making Binary an abstract class, we for simplicity use a class with diverging methods.
aspect Exp_Known {
    introduction Power {
        int raise_3(int base) {
            return this.op.e(base,this.op.e(base,this.op.e(base,this.neutral)));
        }
    }
}

aspect Exp_Op_Neutral_Known {
    introduction Power {
        int raise_3_Mul_1(int base) { return base*(base*(base*1)); }
    }
}

aspect Base_Known {
    introduction Power {
        int raise_2() { return this.loop_2(this.exp); }
        int loop_2(int x) {
            return x==0 ? this.neutral : this.op.e_2(this.loop(x-1));
        }
        introduction Binary { int e_2( int y ) { return this.e_2( y ); } }
        introduction Add { int e_2( int y ) { return 2+y; } }
        introduction Mul { int e_2( int y ) { return 2*y; } }
    }
}

Fig. 2. Various specializations of the power example.

directly residualized. The result is shown in the aspect Exp_Op_Neutral_Known. As a last example, we can specialize the program based only on the information that the base value is known; speculative specialization allows each e method to be specialized for the known base value, as shown in the aspect Base_Known.

3 Extended Featherweight Java

To define partial evaluation for an object-oriented language, we use a small class-based object-oriented language based on Java [11] named Extended Featherweight Java (EFJ) after Featherweight Java [14]. EFJ is intended to constitute a least common denominator for class-based languages so that any partial evaluation principles developed for EFJ will apply to most other class-based languages as well. EFJ is a subset of Java without side-effects, and an EFJ program behaves like the syntactically equivalent Java program. EFJ incorporates classes and inheritance in a statically typed setting, object fields, virtual methods with formal parameters, and object constructors.
Like Java, EFJ is a statically-typed object-oriented language. We will not define the EFJ typing rules here; we refer to the original presentation of Featherweight Java [14] or the author’s PhD dissertation [24] for a description of the EFJ typing rules. Only the subtyping relation between classes is directly used in our formalization; subtyping follows the class hierarchy, and is denoted “<:".

### 3.1 EFJ Syntax

The syntax of EFJ is given in Figure 3. A program is a collection of classes and a main expression. Each class in the program extends some superclass, and declares a number of fields, a constructor, and a number of methods. A constructor always calls the constructor of the superclass first and then initializes each field declared in the class afterward; the constructor is the only place where fields can be assigned values, because there are no side-effects in the language. The definition of a constructor is fixed given the fields of a class and its superclass, and the semantics of object initialization is not defined in terms of the constructor but is defined directly in terms of the fields of the class. However, writing out the constructor allows us to retain a Java-compatible syntax. The body of a method is a single expression. An expression can be a constant, a variable, a field lookup, a virtual method invocation, an object instantiation, a class cast, an operator application, or a conditional.

A value computed by the program can be either a constant or an object; an object is represented as a tuple of values labeled with the name of the class of the object.

The special class Object can neither be declared nor instantiated but is part of every program. This class extends no other class, and has no methods and no fields; with the exception of this class, all classes referenced in the program must also be defined in the program. Furthermore, there should be no cycles in the inheritance relation between classes.
\[\begin{align*}
\sigma \vdash c &\rightarrow c \quad \text{(R-Const)} \\
\sigma \vdash x &\rightarrow \sigma(x) \quad \text{(R-VAR)} \\
\forall i \in 1..n \quad \sigma \vdash e_i &\rightarrow v_i \\
\sigma \vdash \text{new } C(e_1, \ldots, e_n) &\rightarrow \text{object } C(v_1, \ldots, v_n) \quad \text{(R-New)} \\
\sigma \vdash e &\rightarrow \text{object } C(v_1, \ldots, v_n) \\
\quad \text{fields}(C) = T_1 f_1, \ldots, T_n f_n \\
\sigma \vdash e.f_i &\rightarrow v_i \quad \text{(R-Field)} \\
\forall i \in 1..n \quad \sigma \vdash d_i &\rightarrow d'_i \\
\quad mbody(C, m) = ((x_1, \ldots, x_k), e_0) \\
\quad \text{[} x_1 \mapsto d'_1, \ldots, x_k \mapsto d'_k, \text{this } \mapsto \text{object } C(v_1, \ldots, v_n)\text{]} \quad \sigma \vdash e.m(d_1, \ldots, d_k) &\rightarrow v \quad \text{(R-InvK)} \\
\sigma \vdash e &\rightarrow \text{object } C(v_1, \ldots, v_n) \\
\quad C <: D \quad \sigma \vdash (D)e &\rightarrow \text{object } C(v_1, \ldots, v_n) \quad \text{(R-Cast)} \\
\sigma \vdash e_0 &\rightarrow \text{true} \\
\sigma \vdash e_1 &\rightarrow v \quad \sigma \vdash e_0 &\rightarrow \text{false} \\
\sigma \vdash (e_0 ? e_1 : e_2) &\rightarrow v \quad \sigma \vdash e_2 &\rightarrow v \quad \sigma \vdash (e_0 ? e_1 : e_2) &\rightarrow v \quad \text{(R-Cond-T)} \quad \text{(R-Cond-F)} \\
\sigma \vdash e_0 &\rightarrow v_0 \\
\sigma \vdash e_1 &\rightarrow v_1 \quad \Delta_{OP}(v_0, v_1) = v' \quad \sigma \vdash e_0 \text{ OP } e_1 &\rightarrow v' \quad \text{(R-Op)}
\end{align*}\]

Environment \(\sigma : \text{Var} \rightarrow \text{Value}\)

\(\text{fields}(C) = \text{fields of class } C\)

\(\text{mbody}(C, m) = \text{body of method } m \text{ defined in class } C\)

Fig. 4. EFJ computation (see appendix for auxiliary definitions)

### 3.2 EFJ Evaluation

We define EFJ computation using the eager big-step semantics shown in Figure 4. The evaluation rules have the form \(\sigma \vdash e \rightarrow v\), where \(e\) is an expression that is reduced into a value \(v\) in an environment \(\sigma\) that maps variables to values. The evaluation rules are defined as follows. A \textit{new} expression creates an object holding the value of each expression passed to the constructor (\textit{R-New}). A reference to a field retrieves the corresponding value (\textit{R-Field}). Method invocation first reduces the self expression to decide the class of the receiver object, which determines what method is called; the method body is evaluated in an environment that binds the self object to the special variable \textit{this} and binds each formal parameter to the corresponding argument (\textit{R-InvK}). Class casts can only be reduced when the class of the concrete object is a sub-class of the casted type (\textit{R-Cast}). The evaluation rules for the other constructs are straightforward, and will not be discussed. To compute the value of a complete program, the main expression of the program must be evaluated in an environment that defines the values of any free variables in the main expression.
2P ∈ 2Program ::= ({2CL_1,...,2CL_n},2e)
2CL ∈ 2Class ::= class C extends D {T_1 f_1;...;T_n f_n}; 2K 2M1...2M_k
2K ∈ 2Constructor ::= K | K
K ∈ Constructor ::= C(T_1 f_1,...,T_n f_n)
{super(f_1,...,f_j); this.f_1 = f_1;...;this.f_n = f_n}
2M ∈ 2Method ::= T m(2D_1,...,2D_n) {return 2e;}
| T m(2D_1,...,2D_n) {return 2e;}
2D ∈ 2Declaration ::= Tx | Tx
2e ∈ 2Expression ::= e_0 | lift(2e_0) | x | 2e_0 f | 2e_0 m(2e_1,...,2e_n)
| new C(2e_1,...,2e_n) | (C)2e_0 | 2e_0 OP 2e_1 | (2e_0 ?2e_1 :2e_2)
| new C(2e_1,...,2e_n) | (C)2e_0 | 2e_0 OP 2e_1 | (2e_0 ?2e_1 :2e_2)
OP ∈ Operator ::= + | - | * | / | < | > | == | && | ||
c ∈ Constant ::= true | false | 0 | 1 | -1 | ...
C,D ∈ Class-name, f ∈ Field-name, m ∈ Method-name, x ∈ Variable
Values that result from computation:
v ∈ Value ::= c | object_C(v_1,...,v_n) | residual program part

Fig. 5. 2EFJ syntax

4 Two-Level Language

Partial evaluation can be formalized as evaluation in a language with a two-level syntax [15]. The two-level separation of a program corresponds to a division of the program into static and dynamic parts. Since binding times are made syntactically explicit, specialization can be expressed straightforwardly using evaluation rules for the two-level syntax. We use this approach to formalize EFJ specialization.

We extend EFJ into a two-level language by adding new constructs that represent dynamic program parts, as shown in Figure 5; we name this language Two-Level EFJ (2EFJ). Static 2EFJ constructs are written as their EFJ counterparts, whereas dynamic constructs are underlined. Evaluation of a dynamic program part residualizes a specialized program part, so the domain of values is extended to include residual program parts.

To permit a static expression to appear within a dynamic context, we add a lift expression. As is normally the case in partial evaluation, we only allow base-type values to be lifted. Lifting object values could be done by generating residual new-expressions, but doing so would duplicate computation, and would furthermore be problematic in most object-oriented languages since object identity would not be preserved.

We use monovariant binding times, which means that there is exactly one binding time associated with each program point, and that we assign the same binding time to all instances of a given class. Furthermore, we do not allow partially static data, so all fields of a given class have the same binding time. (We return to these restrictions in Section 9.) We indicate the binding time of the objects of a given class by a binding-time annotation on the constructor of
the class. We refer to a class with a statically-annotated constructor as a static class, and similarly for a class with a dynamically-annotated constructor.

For a method definition, the binding-time annotation on the class indicates the binding time of the self, the binding time of each formal parameter is indicated by its annotation, and the annotation on the return keyword indicates the binding time of the method return value.

5 Well-Annotatedness

We now define a set of rules that ensure 2EFJ well-annotatedness: a well-annotated (and well-typed) 2EFJ program either diverges, stops at the reduction of an illegal type cast, or reduces to a specialized program. In this section, we first discuss the relation between a class and its subclasses in terms of binding times, and then give a type system that defines well-annotatedness for 2EFJ programs.

5.1 Binding Times and Inheritance

The binding times of the classes of a program are influenced not only by how object instances are used in the program, but also by the inheritance relation between the classes.

The binding time of two objects that are used at the same program point (a field lookup or method invocation expression) must be equal. We use monovariant binding times, so the classes of such two objects must have the same binding time.

We use a type inferencing algorithm to predict the types of the objects that may be used at a given field access or method invocation, and thereby also to predict the control flow of the program. (Thus, the type inferencing algorithm can also be thought of as a control-flow analysis.) This type inferencing algorithm could in principle infer concrete types (i.e., a type more specific than the qualifying type given in the program); the more precise the type inferencing algorithm, the smaller the set of types at each program point, and thus the fewer restrictions there are on the binding time of each class. For simplicity, we compute the set of types using the EFJ type inference rules: for a given field access or method invocation, the set of possible types is the complete set of subtypes of the type inferred for the expression. Thus, a class that is used as the qualifying type of the self object in a field access or method invocation has the same binding time as its subclasses. Note that the class Object has neither fields nor methods, and thus never serves as the qualifying type in such expressions. More precise type annotations can be obtained by using a more precise type-inference algorithm, several of which are presented in literature [20,21,22].

In summary, the binding times of two classes are linked across a common superclass if an object qualified by this common superclass is the subject of a field access or method invocation. Had we used a more precise type inferencing algorithm, we would have had a different behavior.
5.2 Well-Annottedness Rules

We define well-annotatedness of a 2EFJ program using the rules of Figures 6 and 7. These rules are used to check that the binding-time annotation of each construct in the program is consistent with the annotations on the rest of the program. The rules have the form $\Gamma \vdash e : T$, meaning “in the environment $\Gamma$, the two-level expression $e$ has binding time $T.”” The well-annotatedness rules are syntax directed, and use a number of auxiliary definitions; these definitions are
Methods:

\[\text{bt-signature}(C, m) = T_0, (T_1, \ldots, T_n) \mapsto S\]
\[
\Gamma = \text{build-env}(\text{no-bt}(P), T_0, (T_1, \ldots, T_n)) \quad \Gamma \vdash e : S
\]
\[\text{T m}(P) \{ \text{return } e; \} \text{ OK in } C\]  \(\text{(W-S-METHOD)}\)

\[\text{bt-signature}(C, m) = T_0, (T_1, \ldots, T_n) \mapsto D\]
\[
\Gamma = \text{build-env}(\text{no-bt}(P), T_0, (T_1, \ldots, T_n)) \quad \Gamma \vdash e : D
\]
\[\text{T m}(P) \{ \text{return } e; \} \text{ OK in } C\]  \(\text{(W-D-METHOD)}\)

Classes:

\[
\forall i \in 1 \ldots p \quad M_i \text{ OK in } C
\]
\[\text{class } C \text{ extends } D \{ C_1 f_1; \ldots; C_n f_n \} \text{ OK}\]

Program:

\[
\forall \text{CL} \in \{ \text{CL}_1, \ldots, \text{CL}_n \} : \text{CL} \text{ OK} \quad \Gamma_0 \vdash e : T
\]
\[\{ \{\text{CL}_1, \ldots, \text{CL}_n \}, e \} \text{ OK in } \Gamma_0\]

no-bt: maps a 2EFJ program part into the corresponding EFJ program part
build-env: builds a binding-time environment from a list of formal parameters
and a list of binding times to associate with these parameters

Fig. 7. Rules for well-annotated methods, classes and programs

summarized in the figure, and described in detail in the appendix. We use \(D\) to
indicate a dynamic binding time and \(S\) to indicate a static binding time.

The well-annotatedness rules for expressions (Figure 6) are defined as follows.
The binding-time annotation of a lift expression is dynamic and its argument
must be a static base-type value (W-Lift). The binding-time annotation of a field
access must correspond to the binding time of the field across all classes that
may be used at this program point and is equal to the binding time of the
classes that contain the field (W-S-Field and W-D-Field). The binding time of an
object instantiation must be equal to the binding time of the class that is being
instantiated (W-S-New and W-D-New). Similarly, the binding time of a cast must
be equal to the binding time of the argument and the class that it is being
cast to (W-S-Cast and W-D-Cast). For a method invocation, the binding time of
the self object must be equal to the binding time of the classes of the possible
receiver objects, and the binding times of the parameters must be equal to the
binding times of the actual arguments. The well-annotatedness rules for the other
constructs are straightforward, and will not be discussed.

For a method declaration in a class \(C\) to be well-annotated, the binding time
of its body must be equal to the binding-time annotation on the return statement
(Figure 7, judgment “m OK in C”). The binding time of the body is checked using
the well-annotatedness rules for expressions, in an environment defined by the
binding-time annotations on the class and the method formal parameters. For
a class \(C\) to be well-annotated, each method must be well-annotated (judgment
“C OK”). Similarly, for a program \(P\) to be well-annotated in an environment \(\Gamma_0\)
that provides binding times for any free variables, the main expression and each class must be well-annotated (judgment “P OK in Γ₀”).

6 Specialization

The static parts of a 2EFJ program can be reduced away, leaving behind only the dynamic parts. Evaluation of a well-annotated 2EFJ program either diverges, stops at an illegal static type cast, or results in a specialized program.

6.1 2EFJ Expression Evaluation

Figure 8 shows the definition of 2EFJ expression evaluation. The evaluation rules have the form \( \sigma, P \vdash e \Rightarrow (e', M) \), where \( \sigma \) is an environment that maps variables to values (which may be evaluated program parts), \( P \) is a set of pending methods (methods that are currently being specialized), \( e \) is an expression that is specialized into \( e' \), and \( M \) is a set of new methods generated by the specialization of \( e \).

The static parts of a 2EFJ expression reduce into values using a set of rules that are counterparts to the standard EFJ evaluation rules of Figure 4, extended to pass the set of pending methods inwards and collect specialized methods. For example, the rule \( \text{R-Cast} \) of Figure 4 becomes

\[
\frac{\sigma, P \vdash e \Rightarrow \langle \text{object}_C(v_1, \ldots, v_n), M \rangle}{\sigma, P \vdash (D)e \Rightarrow \langle \text{object}_C(v_1, \ldots, v_n), M \rangle}
\] (2RS-Cast)

The 2EFJ counterpart of an EFJ evaluation rule \( \text{R-x} \) is named \( 2\text{RS-x} \) (Reduce Static), giving the rules (2RS-Const), (2RS-Var), (2RS-New), (2RS-Field), (2RS-Invk), (2RS-Cast), (2RS-OP), and (2RS-Cond). The evaluation rules for static 2EFJ expressions are straightforward except for method invocations; the rule is basically unchanged, but is used differently. A method invocation with a static self object but a dynamic return value will produce a residual expression that is unfolded into the calling context; any arguments, be they values or residual program parts, are substituted throughout the body of the method (2RS-Invk). Note that since methods only are unfolded when the self is static, the unfolded body will contain no references to the fields of the self, and encapsulation is thus preserved.

With the exception of method invocation, all evaluation rules for dynamic constructs are straightforward: each sub-component is reduced into a residual expression, and used to rebuild the construct. There are two rules for evaluating dynamic methods invocations. The first rule (2RD-Invk-Memo) handles the case where a specialized method that can be re-used is in the process of being generated, meaning that it is contained in the set of pending methods. In this case, a call to this method is simply residualized. The function \( X \) used in the definition of this rule evaluates each argument and the self, and collects specialized methods generated by this evaluation. In addition, it determines the indices of those formal parameters that have a static binding-time annotation, and those that have dynamic binding-time annotation. The second rule (2RD-Invk-New) handles
\[
\sigma, P \vdash e \rightarrow \langle v, M \rangle
\]
\[
\text{c = residualize(v)}
\]
\[
\sigma, P \vdash \text{lift}(e) \rightarrow \langle c, M \rangle \quad \text{(2R-LIFT)}
\]
\[
\sigma, P \vdash x \rightarrow \langle x, \emptyset \rangle \quad \text{(2RD-VAR)}
\]
\[
\forall i \in 1 \ldots n \quad \sigma, P \vdash e_i \rightarrow \langle e'_i, M_i \rangle \quad M' = \bigcup_i M_i
\]
\[
\text{new \ C(e_1, \ldots, e_n) \rightarrow \langle \text{new} \ C(e'_1, \ldots, e'_n), M' \rangle} \quad \text{(2RD-NEW)}
\]
\[
\sigma, P \vdash e \rightarrow \langle e', M \rangle \quad \sigma, P \vdash e.f \rightarrow \langle e', M \rangle \quad \text{(2RD-FIELD)}
\]
\[
\sigma, P \vdash e \rightarrow \langle e', M \rangle \quad \sigma, P \vdash (C)e \rightarrow \langle (C)e', M \rangle \quad \text{(2RD-CAST)}
\]
\[
\forall i \in \{0, 1, 2\} \quad \sigma, P \vdash e_i \rightarrow \langle e'_i, M_i \rangle \quad M' = \bigcup_i M_i
\]
\[
\sigma, P \vdash (e_0.0p \ e_1) \rightarrow \langle e'_1.0p \ e'_2, M' \rangle \quad \text{(2RD-OP)}
\]
\[
X(\sigma, P, \text{e.m(d}_1, \ldots, d_k)) = ([d'_1, \ldots, d'_k], C, I_S, I_D, e', M)
\]
\[
(C, m, [d'_i | i \in I_S], m') \in P \quad [p_1, \ldots, p_a] = I_D
\]
\[
\sigma, P \vdash \text{e.m(d}_1, \ldots, d_k) \rightarrow \langle e'.m'(d'_{p_1}, \ldots, d'_{p_a}), M \rangle \quad \text{(2RD-INVK-MEMO)}
\]
\[
X(\sigma, P, \text{e.m(d}_1, \ldots, d_k)) = ([d'_1, \ldots, d'_k], C, I_S, I_D, e', M)
\]
\[
\neg \exists m' : (C, m, [d'_i | i \in I_S], m') \in P \quad G(C, m, I_S, I_D, [d'_i | i \in I_S], P) = (m'', M') \quad [p_1, \ldots, p_a] = I_D
\]
\[
\sigma, P \vdash \text{e.m(d}_1, \ldots, d_k) \rightarrow \langle e'.m''(d'_{p_1}, \ldots, d'_{p_a}), M \cup M' \rangle \quad \text{2RD-INVK-NEW}
\]

- Pending methods \(P\): \(\{C_1, \ldots, C_q\}, m, [e_1, \ldots, e_k], m'\)
- Methods produced \(M\): \((C, M)\)
- \(\text{residualize(v)}\) = residual representation of base-type value \(v\)
- Function \(X\): Given \((\sigma, P, \text{e.m(d}_1, \ldots, d_k))\), return \((E, C, I_S, I_D, e', M)\), where the list \(E\) contains the arguments \((d_1, \ldots, d_k)\) evaluated, \(C\) is the set of possible classes of \(e\), the list \(I_S\) contains the indices of the static formal parameters of \(m\), the list \(I_D\) contains the indices of the dynamic formal parameters of \(m\), \(e'\) is evaluated, and \(M\) is the set of new specialized methods generated by evaluation of \(e\) and \((d_1, \ldots, d_k)\).
- Function \(G\): Given arguments as in rule \((\text{RS-INVK-NEW})\), return \((m'', M)\), where \(m''\) is the name of a new, specialized method, and \(M\) contains all specialized versions of \(m''\) together with any specialized methods generated while specializing \(m''\).

The auxiliary functions \(X\) and \(G\) are defined in Figure 9.

List comprehension notation: \([x_i | i \in L]\) = list containing those \(x_i\) for which \(i \in L\), ordered as in \(L\)

**Fig. 8.** Specialization of dynamic 2EFJ expressions
∀i ∈ 1...k σ, P ⊢ di → ⟨d′i, Mi⟩ M′ = ∪i Mi type(e) = {C1,...,Cq}

\[ X(σ, P, e; m(d1,...,dk)) = ([d′1,...,d′k],\{C1,...,Cq\}, IS, ID, e', M' ∪ M'') \]

\[ m'' = \text{gensym}(m) \quad P'' = \{((C1,...,Cq),m,[d′1,...,d′k],m'')\} \quad T = \text{return-type}(C1,m) \]

\[ [p1,...,pa] = ID \quad \forall j ∈ 1...q \quad \text{mbody}(Cj,m) = ([x′1,...,x′k],ej) \quad \sigma'_j = \{x'_i → d'_i | i ∈ IS\} \]

\[ \sigma'_j, P ∪ P' ⊢ e_j → ⟨e'_j,M_j⟩ \quad m_j = (Cj,T m''(x'p1,...,x'pa)\{\text{return } e'_j\};}) \]

\[ M' = ∪j M_j \]

\[ G(C1,...,Cq,m,IS,ID,[d′1,...,d′k],P') = (m'',M' ∪ \{m1,...,mq\}) \]

\[ S\text{-indices}(C,m)=\text{list of indices of static formal parameters of method } m \text{ of class } C \]

\[ D\text{-indices}(C,m)=\text{list of indices of dynamic formal parameters of method } m \text{ of class } C \]

\[ \text{gensym}(m)=\text{uniquely generated method name based on } m \]

\[ \text{return-type}(C,m)=\text{return type of method } m \text{ in class } C \]

**Fig. 9.** Auxiliary definitions for Figure 8

the case where a set of new specialized methods must be generated. A set of specialized methods all of the same name are generated using the function \( G \), and an invocation of a method of this name is residualized.

The function \( G \) generates a new function name, and uses it together with the static evaluated arguments to extend the set of pending methods. The body of each potential receiver method is determined, and an environment that maps the static formal parameters to the corresponding arguments is constructed for each method. Each body is then evaluated, and used to construct a member in the set of specialized methods.

The evaluation rules for method invocation can be improved in a number of ways. Let-blocks can be used to avoid code duplication when methods with dynamic formal parameters are unfolded (although let-blocks would have to be added to the language), and a cache can be introduced to avoid generating duplicate specialized methods. These problems and their solution are well-known from partial evaluation for functional languages, and will not be discussed.

### 6.2 Evaluation of a Program

Evaluation of a 2EFJ program produces a specialized main expression and a collection of specialized methods; this representation can be transformed into the aspect syntax of Figure 10a. We use *introduction* blocks to introduce specialized methods into classes, and a special *main* block to replace the main expression of a program. The rules for transforming the tuple resulting from 2EFJ evaluation into an aspect are shown in Figure 10b. The aspect produced by specialization can be woven into the main program using a simple weaver *weave*, defined by the evaluation rule of Figure 10c. The overall effect is that each specialized method
Partial Evaluation for Class-Based Object-Oriented Languages

\[ A \in \text{Aspect} ::= \text{aspect } N \{ I_1 \ldots I_n \text{ main } e \} \]
\[ I \in \text{Introduction} ::= \text{introduction } C \{ M_1 \ldots M_n \} \]
\[ N \in \text{Aspect-name} \]

(a) Aspect syntax for program with main expression

\[ \sigma_0, \emptyset \vdash e \implies (e', M) \]
\[ CL_i = \text{class } C_i \text{ extends } D_i \{ \ldots ; K M_1 \ldots M_k \} \quad \forall i \in 1 \ldots n \]
\[ \{ M'_1, \ldots, M'_k \} = \{ m | (C_i, m) \in M \} \]
\[ I_i = \text{introduction } C_i \{ M'_1 \ldots M'_k \} \]

\[ \sigma_0 \vdash (\{ CL_1, \ldots, CL_n \}, e) \implies \text{aspect } \{ I_1 \ldots I_n \text{ main } e' \} \]

(b) Specialization into an aspect

\[ I_i = \text{introduction } C_i \{ M'_1 \ldots M'_k \} \]
\[ CL_i = \text{class } C_i \text{ extends } D_i \{ \ldots ; K M_1 \ldots M_p \} \]
\[ \forall i \in 1 \ldots n \]
\[ CL'_i = \text{class } C_i \text{ extends } D_i \{ \ldots ; K M_1 \ldots M_p M'_1 \ldots M'_k \} \]
\[ \forall i \in 1 \ldots n \]

\[ \text{weave}((\{ CL_1, \ldots, CL_n \}, e), \text{aspect } \{ I_1 \ldots I_n \text{ main } e' \}) \implies (\{ CL'_1, \ldots, CL'_n \}, e') \]

(c) Weaving of aspect and program

Fig. 10. Specialization of a program into an aspect

is inserted into the class for which it was specialized, and that the generic main expression is replaced by the specialized main expression.

7 Binding-Time Analysis

Binding-time analysis of an EFJ program constructs a well-annotated 2EFJ program. The binding-time analysis is supplied the binding times of the free variables of the main expression, and the derived annotations must respect the well-annotatedness rules and should make static as much of the program as is possible. We express the binding-time analysis as constraints on the binding times of the program, and then use a constraint solver to find a consistent solution that assigns binding times to the program.

7.1 Constraint System

We generate one or more constraints for every program part. Constraint variables are associated with expressions, classes, method returns, method formal parameters, and the free variables of the program main expression. A constraint variable \( T_e \) constrains the binding time of an expression \( e \), \( T_C \) constrains the binding time of a class \( C \), and for a method \( m \) in the class \( C \) with formal parameters \( x_1, \ldots, x_n \), \( T_{C,m,x} \) constrains the binding time of each method parameter and \( T_{C,m,\text{return}} \) constrains the binding time of the method return value (the binding time of the self argument is constrained by \( T_C \)). The constraint variable \( T_{\Box,\Box,\Box} \) constrains a free variable \( x \) of the main expression of the program. Apart from equality between binding times, we use the operator \( < \) to constrain a liftable expression, and the operator \( T_1 > T_2 \) to express a dependency between \( T_1 \) and
\[
\begin{align*}
\mathcal{C}^E(C, m, e) &= \text{case } e \text{ of} \\
\item [c] &\Rightarrow \{ T_e = S \} \\
\item [x] &\Rightarrow \{ T_x = T_{c.a,x} \} \\
\item \[e_0.f] &\Rightarrow \{ T_{e_0} = T_{e_0, \ast} \} \cup \mathcal{C}^E(C, m, e_0), \text{type}(e) = \{ C_1, \ldots, C_p \} \\
\item \text{new } D(e_1, \ldots, e_n) &\Rightarrow \{ (T_{e_1}, \overline{T_{e_1}}), \overline{T_{e_1}} = T_{e_0}, T_e = T_0 \} \cup \mathcal{C}^E(C, m, e_i), \forall i \in 1 \ldots n \\
\item [(D)e_1] &\Rightarrow \{ T_e = T_{e_0}, T_e = T_b \} \cup \mathcal{C}^E(C, m, e_1) \\
\item e_0 \text{ OP } e_1 &\Rightarrow \{ (T_{e_0}, \overline{T_{e_0}}), \overline{T_{e_0}} = T_e \} \cup \mathcal{C}^E(C, m, e_i), \forall i \in \{0, 1\} \\
\item (e_1, \text{?e}_2: e_3) &\Rightarrow \{ (T_{e_1}, \overline{T_{e_1}}), T_{e_0} \triangleright T_{e_1}, T_{e_0} \triangleright T_{e_2}, \overline{T_{e_1}} = T_{e_2}, T_e = T_{e_1} = T_{e_2} \} \\
&\cup \mathcal{C}^E(C, m, e_i), \forall i \in \{0, 1, 2\} \\
\item e_0.m(d_1, \ldots, d_n) &\Rightarrow \{ L(T_{d_1}, \overline{T_{d_1}}), T_{d_1} = T_{c_{j.n.x_1}, T_{e_0} = T_{c_j}, T_e = T_{c_{j.n.return}} \} \\
&\cup \mathcal{C}^E(C, m, e_0) \cup \mathcal{C}^E(C, m, d_i), \text{type}(e_0) = \{ C_1, \ldots, C_p \}, \forall i \in 1 \ldots n, \forall j \in 1 \ldots p \\
\item C_M(C, m, e) &= \{ L(T_e, \overline{T_e}), T_{c.n.return} = \overline{T_e}, T_e > T_{c.n.return} \} \cup \mathcal{C}^E(C, m, e) \\
\item C^C(\text{class } C \text{ extends } D \{ \ldots : M_1 \ldots M_n \}) &= \bigcup_i \mathcal{C}^M(C, m, e_i), \text{M}_i = T_{m_i(\ldots)} \{ \text{return } e_i ; \} , \forall i \in 1 \ldots n \\
\item C^P(\{ C_{L_1}, \ldots, C_{L_n} \}, e, I) &= \bigcup \mathcal{C}^E(\square, \square, e) \cup (\bigcup_i \mathcal{C}^C(C_{L_i})), \forall i \in 1 \ldots n \\
\text{Constraints: } T_1 > T_2 &\iff (T_1 = D \Rightarrow T_2 = D) \\& S \prec D \\
L(T_e, \overline{T_e}) &= \begin{cases} T_e \leq \overline{T_e}, \text{type}(e) \in \{ \text{int}, \text{bool} \} \\
T_e = \overline{T_e}, \text{otherwise} \\
\end{cases}
\end{align*}
\]

**Fig. 11.** Constraint generation

The solution produced by the C-Mix constraint solver constrains all program parts that need to be annotated dynamic (e.g., \(T_e = D\) in the solution means \(e\) is annotated dynamic); all other program parts are assigned static binding time. A constraint \(T_e \prec \overline{T_e}\) in the solution indicates that a lift should be inserted around the expression \(e\).

### 7.2 Constraint Solving

To efficiently solve the constraint system generated for an EFJ program, we can directly use the constraint solver of the C-Mix partial evaluator for C [12]. We only use the operators =, >, and <, all of which are identical in C-Mix. Solving our constraint system does not generate new forms of binding times, even though the C-Mix constraint solver treats a richer set of binding times. The solution produced by the C-Mix constraint solver constrains all program parts that need to be annotated dynamic (e.g., \(T_e = D\) in the solution means \(e\) is annotated dynamic); all other program parts are assigned static binding time. A constraint \(T_e \prec \overline{T_e}\) in the solution indicates that a lift should be inserted around the expression \(e\).
8 Examples

To illustrate how the partial evaluator presented in the previous sections can specialize object-oriented programs, we apply it to the power example presented in Section 2 and to an example written using the visitor design pattern [10].

8.1 Power

The power example of Section 2 was specialized informally in three different ways (as shown in Figures 1 and 2). The first specialization scenario, with a known exponent and unknown neutral values and binary operator, cannot be reproduced by our partial evaluator since it requires partially static objects. We can however reproduce the second scenario (where the aspect Exp.Op.Neutral_Known shown in Figure 2 was produced). Given the main expression

\[(\text{new Power}(i1, i2, b ? (\text{Binary})\text{new Add()} : (\text{Binary})\text{new Mul() })).\text{raise}(x)\]

the binding-time analysis is done in an initial environment that annotates the free variables \(i1, i2,\) and \(b\) as static and the free variable \(x\) as dynamic. All classes of the program are annotated dynamic, and hence all method invocations and field accesses are specialized away. The result of specializing is simply the aspect

\[\text{aspect Exp.Op.Neutral_Known} \{ \text{main } x*x*x*1 \} \]

In the last scenario (where the aspect Base.Known was produced), the base value (the variable \(x\) in the main expression above) is static, and all other free variables are dynamic. The binding-time analysis derives a program where all classes of the program are annotated dynamic, and the first argument to the \(e\) methods is annotated as static. The result of specializing is equivalent to the aspect Base.Known, except that the body of the method \(\text{raise.2}\) is the specialized main expression.

8.2 Visitor

The visitor design pattern is a way of specifying an operation to be performed on the elements of an object structure externally to the classes that define this structure [10].

A sample implementation of a tree structure and two visitors is shown in Figure 12, using a slightly relaxed syntax. The class Tree is the abstract superclass that defines the interface for accepting visitors (of type TreeVisitor), and it has concrete subclasses Node and Leaf. The visitor CountOcc counts the number of occurrences of a given element in the tree, and the visitor FoldBinOp folds a binary operator (from Figure 1) over the nodes of the tree.

We can specialize the program to a specific visitor type, as follows. We use an initial environment that defines the binding times of the free variables of the main expression shown in Figure 12: the tree \(t\) is dynamic, the booleans \(b1\) and \(b2\) are static, and the integer \(i\) is static. The binding-time analysis infers that
class Tree {
    int accept(TreeVisitor v) { return this.accept(v); }
}
class Node extends Tree {
    Tree left, right;
    Node(Tree x, Tree y) { this.left = x; this.right = y; }
    int accept(TreeVisitor v) { return v.visitNode(this); }
}
class Leaf extends Tree {
    int val;
    Leaf(int x) { this.val = x; }
    int accept(TreeVisitor t) { return t.visitLeaf(this); }
}
class TreeVisitor {
    int visitLeaf(Leaf f) { return this.visitLeaf(f); }
    int visitNode(Node n) { return this.visitNode(n); }
}
class CountOcc extends TreeVisitor {
    int elm;
    CountOcc(int x) { this.elm = x; }
    int visitLeaf(Leaf l) { return this.elm == l.val ? 1 : 0; }
    int visitNode(Node n) {
        return n.left.accept(this) + n.right.accept(this);
    }
}
class FoldBinOp extends TreeVisitor {
    BinOp op;
    FoldBinOp(BinOp x) { this.op = x; }
    int visitLeaf(Leaf l) { return l.val; }
    int visitNode(Node n) {
        return op.e(n.left.accept(this), n.right.accept(this));
    }
}

t.accept( b1 ? (TreeVisitor)new CountOcc(i) :
            (TreeVisitor)new FoldBinOp( b2 ? new Add() :
                                        new Mul())
}

**Fig. 12.** Source code for tree structure and a few visitors

the class Tree and its subclasses are dynamic, that the classes TreeVisitor and Binary and their subclasses are static, and that all return values and operator applications are dynamic. Specializing the program with the variable b1 as true and the variable i as 2 yields the aspect Count2 shown in Figure 13. Conversely, specializing the program with the variables b1 and b2 as false yields the aspect FoldMul (again shown in Figure 13). In both cases the extra virtual dispatch
needed to select the visitor has been removed, and the implementation of the visitor unfolded into the `accept` methods.

9 Scaling Up to Realistic Java Programs

The EFJ partial evaluator presented in this paper can be extended using existing techniques to specialize realistic Java programs in a useful way. In this section, we first discuss the needed extensions, and then describe a complete partial evaluator for Java implemented according to these guidelines.

9.1 Improving the EFJ Partial Evaluator

Perhaps the most obvious extension to the partial evaluator is the treatment of side-effects. Such an extension is straightforward, since existing techniques from first-order imperative programs (such as the C language) can be used: in our analysis and specialization framework we essentially treat a virtual dispatch as a conditional that selects the receiver method based on the type of the receiver object, and thus no higher-order functions are needed. The challenge lies not in dealing with side-effects, but in defining a binding-time analysis that is sufficiently precise for specializing realistic programs.

To scale up partial evaluation to realistic programs, we must take into account the patterns of programming often found in object-oriented languages. In
a large program, different instances of the same class are often used for different purposes, and will thus often need to be assigned different binding times. Thus, each instance of a given class must be assigned a binding time independently of the other instances of this class. Since an object is usually manipulated through its methods, each invocation of these methods must also be assigned binding times individually. Thus, both class-polyvariance (individual treatment of the instances of each class) and method-polyvariance are needed. In a language with constructors, these must be treated polyvariantly as well. In a language with side-effects, an alias analysis is needed to determine the set of locations manipulated at each program point. The precision of the alias analysis must match the precision of the binding-time analysis, which means that it too needs to be class-polyvariant and method-polyvariant. Furthermore, it is essential to permit partially static objects, which for example can be done with use-sensitivity [13].

There are no formalizations of class and method-polyvariant binding-time analysis with use-sensitive binding times for languages with side-effects. Nevertheless, such a binding-time analysis has been implemented in the Tempo partial evaluator for the C language [7], and is as such “known technology” (the binding-time is polyvariant across C structure instances, which is equivalent to class polyvariance). Polyvariant alias analyses have been studied for imperative languages both formally and in practice, and are well-documented in literature [12].

9.2 Partial Evaluation for Java

We have implemented a complete partial evaluator for Java; this partial evaluator is based on the principles presented in this paper, extended (mostly) as described in Section 9.1 to support larger and more realistic programs [24, 25, 26]. Our partial evaluator, named JSpec, treats the entire Java language excluding exception handlers, although with restrictions on the more exotic features of Java, such as multi-threading and dynamic loading. JSpec has been shown to give significant speedups when applied to large programs written in Java, across different machine architectures and execution environments [24, 25]. However, specialization of large programs written in Java is difficult in practice, due to the complexity of obtaining a satisfactory binding-time division; we are looking to specialization classes [29] and specialization patterns [27] to help guide the specialization process.

10 Related Work

Marquard and Steensgaard have demonstrated the feasibility of on-line partial evaluation for object-oriented languages [18]. They developed a partial evaluator for a small object-based object-oriented language based on Emerald. However, the primary focus is on issues in on-line partial evaluation, such as termination and resource consumption during specialization. There is no description of how partial evaluation should specialize an object-oriented program, and virtually
Partial evaluation can be done based on constructor parameters at run time for C++ programs, as shown by Fujinami [9]. Annotations are used to indicate member methods that are to be specialized. A method is specialized using standard partial evaluation techniques for C and by replacing virtual dispatches through static object references by direct method invocations. Furthermore, if such a method has been tagged as \texttt{inline}, it is inlined into the caller method and further specialized. This approach to partial evaluation for an object-oriented language concentrates on specializing individual objects. On the contrary, we specialize the interaction that takes place between multiple objects based on their respective state, resulting in global specialization of the program.

Veldhuizen has demonstrated that templates in C++ can be used to perform partial evaluation at compile time [28]. By using a combination of template parameters and C++ \texttt{const} constant declarations, arbitrary computations over base type values can be performed at compile time. Nevertheless, specialization with C++ templates is limited in a number of ways: the values that can be manipulated are restricted, the computations that can be simplified are limited, and an explicit two-level syntax must be used to write programs. As a consequence of this last limitation, binding-time analysis must be performed manually, and functionality must be implemented twice if both a generic and a specialized behavior is needed.

Customization and selective argument specialization are highly aggressive yet general-purpose object-oriented compiler optimizations [5,8]. Selective argument specialization (the more general of the two optimization techniques) specializes methods to known type information about their arguments. Specialization is done by eliminating virtual dispatches over objects with known types, similar to partial evaluation. However, there is no dependence on static information, since type information and execution time information is dynamically gathered to control optimizations. Compared to these optimizations, partial evaluation for object-oriented languages is more thorough and more aggressive, but also less general: it propagates values of any type globally throughout the program and reduces any computation that depends only on known information, but does not optimize program parts where no static information is available. In fact, customization and selective argument specialization are often complementary to partial evaluation, since they can be used to optimize program parts of a more dynamic nature, where no static information is known.

11 Conclusion and Future Work

Given the widespread popularity of object-oriented languages and the performance problems associated with frequent use of object-oriented abstractions, we expect that partial evaluation can be a useful software engineering tool when implementing object-oriented software. In this paper, we have given a formal definition of partial evaluation for a minimal class-based object-oriented lan-
guage, and thus made clear how partial evaluation can specialize object-oriented programs. Furthermore, we have described how this minimal partial evaluator can be extended using known partial evaluation techniques to specialize realistic object-oriented programs.

We leave as future work the formal proof of correctness of our partial evaluator. Also, we have concentrated on class-based object-oriented languages. Nonetheless, we consider object-based languages to be an interesting target for partial evaluation, and are working on giving a concise definition of partial evaluation for such languages.

Acknowledgements Thanks are due to Charles Consel and the BRICS programming language group in O2 for feedback on the formalization. I am grateful to the anonymous referees for their perceptive comments, which have greatly improved the quality of this paper, and to Olivier Danvy for providing excellent last-minute advice. Last, special thanks to Julia L. Lawall for her kind help in all phases of writing this paper.

References

10. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.


22. Partial Evaluation and Semantics-Based Program Manipulation (PEPM’93), Copenhagen, Denmark, June 1993. ACM Press.


Class table lookup:

\[
(\{C_1, \ldots, C_n\}, e) \text{ is current program} \\
CCT(C) = \text{class } C \text{ extends } D \{ \ldots \} \in \{C_1, \ldots, C_n\}
\]

Field lookup:

\[
\text{fields}(\text{Object}) = \epsilon \\
CCT(C) = \text{class } C \text{ extends } D \{ \ C_1 f_1; \ldots; C_n f_n \ldots \} \\
\text{fields}(D) = D_1 g_1, \ldots, D_n g_n
\]

Method body lookup:

\[
CCT(C) = \text{class } C \text{ extends } D \{ \ldots M_1 \ldots M_n \} \\
C m(C, x_1, \ldots, x_k) \{ \text{return } e; \} \\
\text{mbody}(m, C) = ((x_1, \ldots, x_k), e) \\
CCT(C) = \text{class } C \text{ extends } D \{ \ldots M_1 \ldots M_n \} \\
\text{m is not defined in } M_1, \ldots, M_n \\
\text{mbody}(m, C) = \text{mbody}(m, D)
\]

Fig. 14. EFJ auxiliary definitions


A Auxiliary Definitions

The function \textit{type} is used throughout the paper to map an expression into a set of types that includes the types of the values that may result from evaluating the expression. To implement this function, we use the EFJ type-inferencing rules [24] in a pre-processing pass, and annotate each expression with its type and (when this type is an object type) all of its sub-types. The 2EFJ evaluation rules exploit the fact that the qualifying type of the expression is included in the set of types returned by the function. If this were not the case, an illegal call to a specialized virtual method might be generated for a dynamic virtual dispatch, since the virtual method must be declared in the qualifying type. Thus, if a more precise type-inference algorithm was used, the qualifying type would have to be explicitly inserted into the set of classes returned by the function.

The definitions in Figure 14 are used to extract information from the program; they are used throughout the paper. The function \textit{CT} maps a class name to its definition, the function \textit{fields} maps a class name to a list of its fields, the function \textit{mbody} maps a method name and a class name to the formal parameters and body of this method. As is the case for the original FJ presentation, we have chosen the notion of a “current program” to avoid threading the program definition through all rules.
no-bt\( (e) = e \) with all binding-time annotations removed

\[
\begin{align*}
\text{build-env}(x_1, \ldots, x_n, T_0, (T_1, \ldots, T_n)) &= [\text{this} \mapsto T_0, x_1 \mapsto T_1, \ldots, x_n \mapsto T_n] \\
\end{align*}
\]

\[
\begin{align*}
CT(C) &= \text{class extends } D \{ \ldots ; K \ldots \} \\
\text{class-bt}(C) &= S \\
\text{class-bt}(C) &= D \\
\text{field-bt}(C, x) &= T \\
\end{align*}
\]

\[
\begin{align*}
brt\_signature(C, m) &= T_0'((T_1', \ldots, T_n') \mapsto \text{param-bt}(\#i(P))) \\
\\
M_i = C.m(P) \{ \text{return } e \} \quad T_0' = \text{class-bt}(C) \quad \forall i \in 1 \ldots \text{arity}(C, m) \quad T_i' = \text{param-bt}(\#i(P)) \\
\\
\text{S-indices}(C, m) &= [i \in 1 \ldots \text{arity}(C, m) \mid \text{param-bt}(\#i(P)) = S] \\
\text{mbody}(C, m) &= (P, e) \\
\text{D-indices}(C, m) &= [i \in 1 \ldots \text{arity}(C, m) \mid \text{param-bt}(\#i(P)) = D] \\
\text{param-bt}(x) &= S \quad \text{param-bt}(x, T) = D \\
\text{mbody}(C, m) &= (P, e) \\
(x_1, \ldots, x_n) &= \text{no-bt}(P) \\
\text{arity}(C, m) &= n
\end{align*}
\]

Fig. 15. Auxiliary definitions for Figures 6, 7 and 9

Figure 15 defines the auxiliary definitions used in Figures 6, 7, and 9. The function no-bt removes all binding-time annotations from an expression. The function build-env builds a type environment for analysis of a method. The function class-bt returns the binding-time of a class, and the function field-bt returns the binding-time of a given field of a class. The function brt-signature returns the binding-time signature of a method. The functions S-indices and D-indices return lists of indices of method formal parameters that have static (S-indices) and dynamic (D-indices) binding time. Last, the functions param-bt and arity are auxiliary function used in this figure.
Driving in the Jungle

Jens Peter Secher

Department of Computer Science, University of Copenhagen (DIKU),
Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark.
jpsecher@diku.dk

Abstract. Collapsed jungle evaluation is an evaluation strategy for functional programs that can give super-linear speedups compared to conventional evaluation strategies such as call-by-need. However, the former strategy may incur administrative evaluation overhead. We demonstrate how this overhead can be eliminated by transforming the program using a variation of positive supercompilation in which the transformation strategy is based on collapsed jungle evaluation. In penetrating the constant-factor barrier, we seem to be close to establishing a transformation technique that guarantees the efficiency of the transformed program. As a spin-off, we clarify the relationship between call-by-name, call-by-need and collapsed-jungle evaluation, showing that all three can be expressed as instances of a common semantics in which the variations — differing only in efficiency — are obtained by varying the degree of sharing in a DAG representation.

1 Introduction

Jungle evaluation has arisen from the graph grammar community as a means of speeding up evaluation of term rewrite systems, as described by Habel, Hoffmann, Kreowski and Plump [7, 6]. In short, a jungle is a directed acyclic graph with explicit addresses of nodes. It has been shown that the naive implementation of a function calculating Fibonacci numbers can be made to run in linear time by using evaluation on fully-collapsed jungles [7]. This kind of evaluation is achieved by never allocating new vertices if identical vertices are present in the graph.

The fully-collapsed-jungle approach has the drawback that it can be somewhat expensive to administer the graph. In this paper we will show how we can remove the run-time overhead of the above implementation technique by shifting the use of fully-collapsed jungles from run time to compile time. Specifically, we will do program transformation on fully-collapsed jungles instead of trees. The result is that we pay for the overhead once and for all, not every time a program is executed.

A spin-off of this approach is that we present a unified formalism for graph reduction, encompassing call-by-name, call-by-need, and collapsed-jungle evaluation (of which only the former has been omitted in this paper due to space restrictions). We will show that these three reduction strategies can be captured
in full by two abstract operations on graphs. We will do this by presenting a programming-language semantics parameterised over these two abstract operations; we will show that any implementation of these two abstract operations will result in the same semantics, as long as the implementation fulfils reasonable criteria. The distinction between any two implementations of the abstract operations is then only one of efficiency. This clarification seems interesting in its own right, since it makes it possible to compare variations of graph reduction.

2 Notation

We will use a notation that is somewhat non-standard, and an explanation is thus in order. When we write, say “abc d”, you should read this as ((ab)c)d. If a is a function, then abc d means the result (if any) of ((ab)c)d. If a is an uninterpreted symbol (a constructor), then such an application means the term (i.e., tree) consisting a root labelled a and ordered children b, c, and d.

We let \{abc d\} denote the set containing the objects a, b, c, and d; we let ⟨abc d⟩ denote the tuple containing these four objects; and we let [abc d] denote the list containing these objects. We use parentheses only as meta-syntax to group objects, that is, to avoid ambiguous interpretations. Thus \{(abc d)\} denotes a singleton set (the element being whatever abc d means). We use ∪, +, and \ as infix operators on sets to denote union, disjoint union, and subtraction, respectively. Given a set S, the power set of S is denoted \mathcal{P}(S); the set of finite lists of elements from S is denoted S*.

We will often need to write sequences such as x1 x2 x3 x4 x5, and we will therefore introduce the shorthand notation (x·)5 for such a sequence: The superscript “5” denotes that the preceding syntactic object “x” should be replicated five times, with the dot replaced by the consecutive numbers 1, 2, 3, 4, 5. If the replicated object is syntactically simple, we will leave out the dot altogether, and, for example, write xn instead of (x·)n. When this kind of notation is used in several layers, the innermost part is expanded first. Hence \{(x. ↦ t2)n\} means \{(x1 ↦ t1 t2) · · · (xn−1 ↦ t1 t2) (xn ↦ t1 t2)\}; the empty sequence is also allowed, so \{(x. ↦ t.)0\} means \{} = ∅.

For a relation → ⊆ S × T, we define the domain \mathcal{D}(→) \triangleq \{s | ∃t : s → t\}. We say that → is deterministic if, for all s ∈ S, s → t and s → t‘ imply t = t‘, that is, → is a (partial) function. To denote that f is a (partial) function, we write f ∈ S → T. If the domain of f is finite, we will use \{(s. ↦ t.)n\} to denote the set of bindings that f comprises. If → is a binary relation S × S, we denote by →+ the transitive closure of →, and by →∗ the reflexive closure of →+. The normal forms of → is the set \mathcal{E}(→) \triangleq S \setminus \mathcal{D}(→).

3 Subject Language

Our programming language is a small, non-strict, first-order, function-oriented language with structured data and pattern matching.
**Example 1 (Fibonacci Numbers).** The following program defines the well-known Fibonacci function:

```
data Nat = 0 | s Nat
fib 0 = s 0
fib (s x) = aux x
aux 0 = s 0
aux (s y) = add (aux y) (fib y)
```

We have

```
add 0 y = y
add (s x) y = s (add x y)
```

**Remark 2.** The above program contains a data-type definition. For clarity, we will put such data-type definitions in our example programs, even though such data-type definitions are not permitted in the language.

### 3.1 Syntax

**Definition 3.** Assume denumerable disjoint sets of symbols for constructors $\mathcal{C}$, functions $\mathcal{F}$, pattern functions $\mathcal{G}$ (ranged over by $c$, $f$, and $g$, respectively), and variables $\mathcal{X}$ (ranged over by $x$, $y$). Then the set of programs $\mathcal{Q}$, definitions $\mathcal{D}$, terms $\mathcal{T}$, and patterns $\mathcal{P}$ (ranged over by $q$, $d$, $t$, and $p$, respectively) are defined by the abstract syntax grammar

- (program) $\mathcal{Q} \ni q ::= d^m$
- (definition) $\mathcal{D} \ni d ::= f \, x^n = t \mid (g \, p \, x^n = t)^m$
- (pattern) $\mathcal{P} \ni p ::= c \, x^n$
- (term) $\mathcal{T} \ni t ::= x \mid c \, t^n \mid f \, t^n \mid g \, t^m$
- (value) $\mathcal{V} \ni v ::= c \, v^n$

where $n \geq 0$ and $m > 0$. We require that

1. No (pattern) function name is defined more than once.
2. No two patterns in a matcher definition contain the same constructor.
3. No variable occurs more than once in the left-hand side of a function definition (the definition is *left-linear*).
4. All variables in the body of a function definition are present among the variables in the left-hand side of the definition.

We let $f \, x^n \Downarrow t$ denote that the program $q$ contains a definition $f \, x^n = t$, and similarly for $g \, p \, x^n \Downarrow t$. As a shorthand, we let $\mathcal{E} = \mathcal{C} \cup \mathcal{F} \cup \mathcal{G}$. The set of variables in a term $t$ is denoted $\mathcal{V}(t)$, a term $t$ is a ground term if $\mathcal{V}(t) = \emptyset$.

We will shortly define the *meaning* of our little language in terms of a (parameterised) small-step operational semantics. The idea is that, given a ground term $t$, we can determine which ground term $t'$ (if any) that $t$ reduces to in one step. The usual way to express such an reduction step is to define a relation on terms via substitutions.

**Definition 4 (Substitution, renaming).** Given a function $\psi = \{ (x. \mapsto t)^n \} \in \mathcal{X} \rightarrow \mathcal{T}$ from variables to terms, we denote by $\theta = \{ (x. \mapsto t)^n \} \in \mathcal{T} \rightarrow \mathcal{T}$ the substitution induced by $\psi$. 

$\square$
3.2 Graphs

In this section we will investigate the interaction between various alternative representations of terms. We therefore formally introduce a more general representation of terms, namely directed acyclic graphs (DAGs).

**Definition 5 (Graphs).** Let \( s \) range over a finite set of symbols \( S \), and let \( \alpha \) and \( \beta \) range over a set of *addresses* \( \mathcal{A} \). We then define the following:

1. We denote by *nodes over \( S \)* the set \( \mathcal{N}(S) \triangleq S \times \mathcal{A}^* \), ranged over by \( \nu \). We will use the shorter notation \( \nu = s\alpha^n \) instead of writing \( \nu = \langle s[\alpha^n] \rangle \).
2. We denote by *directed graphs over \( S \)* the set \( \mathcal{G}(S) \subseteq \mathcal{A} \rightarrow (\mathcal{N}(S) + \mathcal{A}) \) of partial mappings from addresses to nodes/addresses. Any directed graph \( G \) induces a binary relation \( \sim \subseteq \mathcal{A} \times \mathcal{A} \) defined by \( \beta \sim \alpha_i \) iff \( G\beta = \alpha_i \) or \( G\beta = s\alpha^n \land i \leq n \).
3. We denote by *acyclic directed graphs over \( S \)* the set \( \mathcal{G}^*(S) \subseteq \mathcal{A} \rightarrow (\mathcal{N}(S) + \mathcal{A}) \) of acyclic graphs (i.e., \( G \) acyclic iff \( \nexists \alpha \in \mathcal{D}(G) : \alpha \twoheadrightarrow \alpha \)). We let \( \nabla \) range over the set of acyclic graphs.
4. We denote by *configurations over \( S \)* the set \( \mathcal{K}(S) \triangleq \mathcal{G}(S) \times \mathcal{A}, \) ranged over by \( \kappa \).
5. If an address is mapped to another address (i.e., not to a node), we call both the former address and the mapping an *indirection*, and we define

\[
\|\nabla\| \triangleq \{ \alpha \mapsto \beta \mid \nabla \alpha = \beta \in \mathcal{A} \} .
\]

The relation \( \leadsto \subseteq \mathcal{K}(\mathcal{E}) \times \mathcal{A} \) is defined inductively by

\[
\langle \nabla \alpha \rangle \leadsto \alpha , \text{ if } \alpha \notin \mathcal{D}(\|\nabla\|) \\
\langle \nabla \alpha \rangle \leadsto \alpha' , \text{ if } \|\nabla\| = \alpha'' \text{ and } \langle \nabla \alpha'' \rangle \leadsto \alpha' .
\]

6. Let \( \mathcal{A} \) be a set of addresses. We let \( \nabla/\mathcal{A} \) denote the graph \( \nabla \) restricted to \( \mathcal{D}(\nabla) \setminus \mathcal{A} \).
7. We let \( \text{FRESH} \) be a procedure that provides us with a *completely* new address each time it is called. We implicitly assume that every address mentioned has been drawn by this procedure; thus \( \text{FRESH} \) will provide addresses that cannot be captured. \( \square \)

**Example 6 (DAG).** Let \{0 \text{ s add fib aux}\} be a set of symbols and \{\alpha \gamma \delta \beta \epsilon\} a set of addresses, and consider the graph

\[
\nabla = \left\{\begin{array}{c}
\alpha \mapsto \text{add} \\
\gamma \mapsto \text{aux} + \epsilon \\
\delta \mapsto \text{fib} + \epsilon \\
\beta \mapsto \delta
\end{array}\right. .
\]

Then \( \mathcal{D}(\nabla) = \{\alpha \gamma \delta \beta\}, \|\nabla\| = \{\beta \mapsto \delta\}, \langle \nabla \beta \rangle \leadsto \delta \) and e.g., \( \langle \nabla \epsilon \rangle \leadsto \epsilon \). \( \square \)
From the above example, you can see that we intend to use the set \( \mathcal{E} = \mathcal{C} \cup \mathcal{F} \cup \mathcal{G} \) as the set of symbols that our graphs are defined over. We will now clarify the correspondence between such graphs and the terms of our little language. It is fairly straightforward to extract a term from a graph:

**Definition 7 (Extract).** Let \( \phi \in \mathcal{A} \leftrightarrow \mathcal{X} \) be a unique bijection from addresses to variables. By \( \phi_\alpha \) we denote the variable \( \phi \alpha \). The function \( \text{xtract} \in \mathcal{X} (\mathcal{E}) \to \mathcal{T} \) is defined inductively by

\[
\begin{align*}
\text{xtract} \left( \nabla \alpha \right) & \triangleq \phi_\alpha, \text{ if } \alpha \notin \mathcal{D}(\nabla) \\
\text{xtract} \left( (\nabla + \{ \alpha \mapsto \beta \}) \alpha \right) & \triangleq \text{xtract} (\nabla \beta) \\
\text{xtract} \left( (\nabla + \{ \alpha \mapsto s \cdot \beta^n \}) \alpha \right) & \triangleq s \left( \text{xtract} (\nabla \beta) \right)^n 
\end{align*}
\]

\[ {} \]

**Example 8 (Extraction).** Assume \( \phi_\epsilon = x \), and consider the graph \( \nabla \) from Ex.6; we have that \( \text{xtract} (\nabla \beta) = \text{fib} \cdot x \) and \( \text{xtract} (\nabla \alpha) = \text{add} (\text{aux} \cdot x) (\text{fib} \cdot x) \). \( \Box \)

The opposite translation — from terms to graphs — however, depends on how much sharing of sub-terms we want to have. Furthermore, the precise formulation of operations on the graph representation of a term will depend on how far we are willing to go to maintain sharing of sub-terms. To abstract away from such preferences, we define two basic operations on our graphs. The first operation, \( \text{upd} \), has the signature \( \text{upd} \in \mathcal{G} (\mathcal{E}) \to \mathcal{A} \to \mathcal{N} (\mathcal{E}) \to \mathcal{G} (\mathcal{E}) \). It takes as arguments a graph \( \nabla \), a target address \( \alpha \), a node \( \nu \); Intuitively, calling \( \text{upd} \) is like placing a piece in a jigsaw puzzle: the pockets in the new piece are attached to the tabs of the surrounding puzzle, and likewise the tabs of the piece are fitted into the surrounding pockets; more concretely, \( \text{upd} \) updates \( \nabla \) with \( \nu \) at \( \alpha \) such that it connects with existing nodes in \( \nabla \). Formally, it must hold that

\[
\nabla' = \text{upd} \nabla \alpha (s \cdot \alpha^n)
\]

implies

\[
\forall \beta \in \mathcal{A} : \text{xtract} (\nabla' \beta) = (\{ \phi_\alpha \mapsto s \left( \text{xtract} (\nabla \alpha) \right)^n \}) (\text{xtract} (\nabla \beta)) \tag{3}
\]

provided \( \alpha \notin \mathcal{D}(\nabla) \). The above says that we should be able to extract the same terms from the updated graph, except that the variable \( \phi_\alpha \) has been replaced with \( s \left( \text{xtract} (\nabla \alpha) \right)^n \). A straightforward implementation obeying this rule is

\[
\text{upd} \nabla \alpha \nu \triangleq \nabla + \{ \alpha \mapsto \nu \}.
\]

The second operation, \( \text{subst} \), has the signature \( \text{subst} \in \mathcal{G} (\mathcal{E}) \to \mathcal{A} \to \mathcal{T} \to \Psi \to \mathcal{G} (\mathcal{E}) \), where \( \Psi \triangleq \mathcal{X} \to \mathcal{A} \) are partial functions from variables to addresses. Function \( \text{subst} \) takes as arguments a graph \( \nabla \), a target address \( \alpha \), a term \( t \), and a mapping \( \psi \) (from the free variables in \( t \) to addresses in \( \nabla \)). To stick with the jigsaw-puzzle analogy, the effect of \( \text{subst} \) is to cut a picture into a collection of pieces, and then connect this collection of pieces to the existing puzzle; from \( t \), a collection of nodes is made such that the variables of \( t \) are connected to existing nodes, and the root of \( t \) is placed at address \( \alpha \). As the name suggests, this operation is used to perform what corresponds to substitution in the world of terms. Formally, it must hold that
\[ \nabla' = \text{subst} \ \nabla \ \alpha \ t \ \psi \]
implies
\[ \forall \beta \in \mathcal{D}(\nabla) \cup \{\alpha\} : \text{xtract} \ \langle \nabla' \ \beta \rangle = \theta \ (\text{xtract} \ \langle \nabla \ \beta \rangle) \]
where
\[ \theta = \{ \phi_\alpha \mapsto \{ x \mapsto \text{xtract} \ \langle \nabla (\psi \ x) \rangle \mid x \in \mathcal{V}(t) \} \} \, , \]

provided \( \alpha \notin \mathcal{D}(\nabla) \). The above says that we only allow extensions to graphs, that is, we require that, as long as we stay inside the domain of the graph as it were, the same terms will be extracted after the operation, except that, instead of address \( \alpha \) materialising into a variable \( \phi_\alpha \), \( \alpha \) will now materialise into \( t \) with the free variables replaced. Note that the operation might have added more than just \( \alpha \) to the domain of the graph. See Fig. 1 for a straightforward implementation.

**Example 9.** Take the graph \( \nabla \) from Ex. 6 and let \( \nabla_0 = \nabla / \{ \beta \} \), shown below to the left. Two legal results w.r.t. \( \# \) of \( \text{subst} \ \nabla_0 \ \beta (s \ (\text{fib} \ x)) \ \{ x \mapsto \epsilon \} \) are shown below to the right.

In the rest of this section we will be concerned with properties that are independent of the particular implementation of \( \text{upd} \) and \( \text{subst} \). We will therefore talk about families of operations and functions indexed by such implementations.

**Definition 10 (Realisation).** A realisation \( I \) is an implementation of \( \text{upd} \) and \( \text{subst} \), denoted \( \text{upd}_I \) and \( \text{subst}_I \), such that \( \text{upd}_I \) satisfies \( \# \) and \( \text{subst}_I \) satisfies \( \# \).

### 3.3 Semantics

As promised, we can now present a semantics for our little language. The semantics is a small-step operational semantics (Plotkin style [13]) parametrised by a realisation. First we need to translate the initial ground term into graph representation.

**Definition 11 (Initial configuration).** Given a realisation \( I \), the function \( \text{init}_I \in T \rightarrow \mathcal{K}(\mathcal{E}) \) is defined as \( \text{init}_I \ t \triangleq \langle \text{subst}_I \ \emptyset \ \alpha_0 \ \emptyset \ \alpha_0 \rangle \) where \( \alpha_0 = \text{fresh} \).

That is, a new graph is built (on top of an empty graph) such that it represents the initial ground term. Since \( t \) is a ground term, it contains no variables, and thus the variable-to-address mapping is empty.
Theorem 12. For all realisations \( I \) and ground terms \( t \), \( \text{xtract} (\text{init}_I t) = t \).

Definition 13 (\( \rightarrow_{q \cdot I} \)). Given a program \( q \) and a realisation \( I \), the binary relation \( \rightarrow_{q \cdot I} \subseteq \mathcal{K} (\mathcal{E}) \times \mathcal{K} (\mathcal{E}) \) is defined as the smallest relation satisfying the inference system

\[
\frac{f \ x^n \stackrel{q}{\rightarrow} t}{\langle (\nabla + \{ \alpha \mapsto f \cdot \alpha^n \}) \alpha \rangle \rightarrow \langle \nabla' \alpha \rangle}
\]

\[
\frac{\langle \nabla \alpha_0 \rangle \leadsto \alpha'_0}{\langle (\nabla + \{ \alpha \mapsto g \cdot \alpha_0 \alpha^n \}) \alpha \rangle \rightarrow \langle \nabla' \alpha \rangle}
\]

\[
\frac{\langle \nabla \alpha_0 \rangle \leadsto \alpha'_0}{\langle (\nabla + \{ \alpha \mapsto g \cdot \alpha_0 \alpha^n \}) \alpha \rangle \rightarrow \langle \nabla' \alpha \rangle}
\]

\[
\frac{m \leq n \quad \forall i < m : \text{xtract} (\nabla \alpha_i) \in \mathcal{V}}{\langle (\nabla + \{ \alpha \mapsto c \cdot \alpha^n \}) \alpha \rangle \rightarrow \langle \text{upd} (\nabla' \alpha (c \cdot \alpha^n)) \alpha \rangle}
\]

\[
\frac{\langle \nabla \alpha \rangle \leadsto \alpha'}{\langle \nabla \alpha \rangle \rightarrow \langle \nabla' \alpha' \rangle}
\]

The subscript \( q \cdot I \) has been omitted to avoid clutter. \( \square \)

The inference rules define three relations: \( \rightarrow, \Rightarrow, \) and \( \rightarrow_{\alpha} \). The relation \( \rightarrow \) relates a configuration containing a function call to the result of the call. Operationally, you can read the rule apply as “replace the function call with the function body, in which the variables have been replaced by the arguments to the function”; the subst function takes care of both the substitution and the translation from term to graph representation. The select and dive rules take care of pattern-matching functions: In the former case, the first argument to the function has an outermost constructor, and therefore the call can be replaced by the body of the matching function (similar to the apply rule). In the latter case, the first argument to the function does itself contain something that can be rewritten by a single step (i.e., a function call); this one-step rewrite is performed, and the result of this rewrite is written back into the graph by an upd call. The
mutually recursive relations $\leadsto$ and $\rightarrow$ “dig into” the graph to locate the next function call to reduce. The skip rule simply skips over indirections and passes control to the trans or const rules, of which the former simply passes control to the above-mentioned $\leadsto$ relation, which means that a function call has been located. If a function call has not been located — that is, as long as there are only constructors to the left of the current address — the const rule digs into the leftmost subgraph that can be rewritten (i.e., contains a function call); the rewrite is performed, and the result is written back into the graph by an upd call. The xtract $\langle \nabla \alpha_i \rangle \in \mathcal{V}$ part of the premise for const ensures that no rewrites are possible to the left of $\alpha_m$. The relation $\rightarrow$ is thus responsible for reducing the graph from left to right.

**Theorem 14.** The relation $\rightarrow$ is well-defined and deterministic.

**Definition 15 (Evaluation).** Given a realisation $I$, we define the function $\text{eval}_{q,I} : T \rightarrow \mathcal{P}(\mathcal{V})$ by

$$\text{eval}_{q,I} t = \{ v \mid (\text{init}_I t) \star_{q,I} ^* \kappa \in \mathcal{S}(\rightarrow) \land (\text{xtract} \kappa) = v \in \mathcal{V} \}.$$  

We will now state two important properties about the semantics of our little language. The first is a direct consequence of the relation $\rightarrow$ being deterministic.

**Corollary 16.** A ground term evaluates to at most one value.

A ground term may fail to evaluate to a value for two reasons: Either the computation consists of an infinite number of steps, or the computation “gets stuck” at some point. The former reason is usually called non-termination and is an inherent unpleasantness in any universal programming language. We can, however, circumvent the latter situation by imposing a standard polymorphic type system on our language to reject program/term pairs that will get stuck in a normal form that is not a value. We will not pursue this matter further in this paper, but simply assume that all programs and terms are type correct.

**Definition 17 (Correct).** Given program $q$ and ground term $t$, we say that the pair $\langle qt \rangle$ is correct if $(\text{init}_I t) \star_{q,I} ^* \kappa \in \mathcal{S}(\rightarrow)$ implies $(\text{xtract} \kappa) \in \mathcal{V}$, for all realisations $I$.

The second property — and the main reason for the preceding rigor — is that the evaluation of a term (w.r.t. a particular program) always gives us the same result for any realisation.

**Theorem 18 (Realisation independence).** For any program $q$ and ground term $t$, if the pair $\langle qt \rangle$ is correct, then $\text{eval}_{q,A} t = \text{eval}_{q,B} t$ for any two realisations $A$ and $B$.

**Example 19.** Consider the program $q$ in Ex.11. The pair consisting of $q$ and term $\text{fib} (s (s (s (s 0))))$ is correct and evaluates to $s (s (s (s 0))))$. The pair consisting of $q$ and term $\text{fib} (s \lambda)$ is not correct, since evaluation gets stuck in a configuration representing the term aux $\lambda$.  


4 Graph Machinery

In this section we will present two implementations of subst and upd. The two implementations differ in how they handle sharing of identical subterms.

4.1 Call-by-Need

The most straightforward of these implementations is shown in Fig. 1. The explanation of the implementation is as follows.

The upd function simply adds a node to the graph. The subst function calls the auxiliary function saux to ensure that t is converted into graph representation. If the resulting address α′ of this conversion is different from the preferred target α, an indirection is made from α to α′. The auxiliary function saux converts a term t into graph representation such that the variables in t are converted into existing addresses in the graph, thus possibly introducing sharing of subgraphs. More precisely, saux adds new nodes to the graph by recursively decomposing t: If t has s as root and n subterms, n fresh addresses are chosen and fed to recursive calls to saux (thus ensuring that the n subterms have been converted into graph representation), and a new node labelled s is created. If t is a variable x, however, no new node is created; instead the provided mapping ψ tells us which existing address to “substitute” for x. The address representing t can thus be different from the preferred target α.

\[
\text{subst} \in \mathcal{G}(\mathcal{E}) \to \mathcal{A} \to \mathcal{T} \to \Psi \to \mathcal{G}(\mathcal{E})
\]

\[
\text{subst} \triangleq \begin{cases} \text{let } \langle \nabla' \alpha' \rangle = \text{saux } \alpha \psi \text{ in if } \alpha = \alpha' \text{ then } \nabla' \text{ else } \nabla + \{ \alpha \mapsto \alpha' \} \end{cases}
\]

\[
\text{saux} \in \mathcal{G}(\mathcal{E}) \to \mathcal{A} \to \mathcal{T} \to \Psi \to \mathcal{K}(\mathcal{E})
\]

\[
\text{saux} \triangleq \begin{cases} \text{if } t \in \chi \text{ then } \langle \nabla_0(\psi t) \rangle \text{ else let } s t^n = t; (\langle (\nabla. \alpha) \rangle = \text{saux } \nabla -1 \text{ fresh } t. \psi)^n \text{ in } \langle (\nabla_n + \{ \alpha \mapsto s \alpha^n \}) \alpha \rangle \end{cases}
\]

\[
\text{upd} \in \mathcal{G}(\mathcal{E}) \to \mathcal{A} \to \mathcal{N}(\mathcal{E}) \to \mathcal{G}(\mathcal{E})
\]

\[
\text{upd} \triangleq \nabla + \{ \alpha \mapsto \nu \}
\]

Fig. 1. DAG representation of terms (call-by-need)

It is not hard to see that the implementation in Fig. 1 provides the basis for the standard notion of call-by-need: when used in conjunction with the inference rules defined in Def. 13 all occurrences of the same variable (in a function body) share the same subgraph. When it is necessary to reduce one of these occurrences, all the other occurrences will share the rewrite performed by the inference rules.

4.2 Collapsed Jungle

The implementation presented in Fig. 2 is far more interesting, since it will maintain as much sharing as possible. The upd function will never add a new node if
there already exists a similar node. That is, a new node $\nu$ will not be added to the graph $\nabla$ at address $\alpha$ if there already exists a node at $\beta$ such that $\text{xtract} \langle \nabla \beta \rangle = \text{xtract} \langle (\nabla + \{\alpha \mapsto \nu\}) \alpha \rangle$. In case such a $\beta$ exists, we only add an indirection from $\alpha$ to $\beta$ to the graph. Furthermore, after adding something to the graph (be it a node or an indirection), the resulting graph is \textit{collapsed} such that multiple occurrences of similar nodes (in the above sense) are eliminated, and all nodes will have nodes (not indirections) as descendants. Similarly, the $\text{subst}$ function will make sure that superfluous nodes are not added to the graph. $\text{subst}$ will call the auxiliary function $\text{saux}$ to convert the term into graph representation, and if the resulting address is different from the preferred target, an indirection is created, and the (collapsed) graph is returned. The auxiliary function $\text{saux}$ works like in the call-by-need case, except that a node is not created if a similar one exists.

\[
\text{subst} \in \mathcal{G}(\mathcal{E}) \rightarrow \mathcal{A} \rightarrow \mathcal{T} \rightarrow \Psi \rightarrow \mathcal{G}(\mathcal{E})
\]

\[
\text{subst} \nabla \alpha t \psi \triangleq \begin{cases} \text{let} \langle \nabla' \alpha' \rangle = \text{saux} \nabla \alpha t \psi & \text{in if } \alpha' = \alpha \text{ then } \nabla' + \{\alpha \mapsto \alpha'\} \\ \text{else } \text{collapse} (\langle \nabla' \rangle + \{\alpha \mapsto \alpha'\}) & \end{cases}
\]

\[
\text{saux} \in \mathcal{G}(\mathcal{E}) \rightarrow \mathcal{A} \rightarrow \mathcal{T} \rightarrow \Psi \rightarrow \mathcal{G}(\mathcal{E}) \times \mathcal{A}
\]

\[
\text{saux} \nabla_0 \alpha t \psi \triangleq \begin{cases} \text{if } t \in \mathcal{X} \text{ then let } \langle \nabla_0 (\psi t) \rangle/;; \alpha' \text{ in } \langle \nabla_0 \alpha' \rangle & \\ \text{else let } s t^n = t; \langle (\nabla. \alpha.) = \text{saux} \nabla\alpha t \psi \rangle^n & \\ \text{ in if } \exists \beta \in \mathcal{P}(\nabla_n) : \nabla_n \beta = s 4 \beta^n \land (\langle \nabla_n \beta \rangle \leadsto \alpha.)^n & \\ \text{ then collapse } (\langle \nabla_n \beta \rangle + \{\alpha \mapsto s 4 \alpha^n\} \alpha) & \\ \text{else collapse } (\langle \nabla_n \beta \rangle + \{\alpha \mapsto s 4 \alpha^n\} \alpha) & \end{cases}
\]

\[
\text{upd} \ n \in \mathcal{G}(\mathcal{E}) \rightarrow \mathcal{A} \rightarrow \mathcal{N}(\mathcal{E}) \rightarrow \mathcal{G}(\mathcal{E})
\]

\[
\text{upd} \nabla \alpha (s 4 \alpha^n) \triangleq \begin{cases} \text{let } (\langle \nabla. \alpha. \rangle \leadsto \alpha.')^n \text{ in if } \exists \beta : \nabla \beta = s 4 \beta^n \land (\langle \nabla. \beta. \rangle \leadsto \alpha.')^n & \\ \text{ then collapse } (\langle \nabla + \{\alpha \mapsto \beta\} \rangle & \\ \text{else collapse } (\langle \nabla + \{\alpha \mapsto s 4 \alpha^n\} \rangle & \end{cases}
\]

\[
\text{collapse} \ n \in \mathcal{G}(\mathcal{E}) \rightarrow \mathcal{G}(\mathcal{E})
\]

\[
\text{collapse} \nabla \triangleq \begin{cases} \text{if } \exists \alpha : \nabla \alpha = s 4 \alpha^n \land \{\alpha^n\} \cap \mathcal{P}(\{\nabla\}) \neq \emptyset \text{ then } \nabla & \\ \text{else let } \nabla' + \{\alpha \mapsto s 4 \alpha^n\} = \nabla : \{\alpha^n\} \cap \mathcal{P}(\{\nabla\}) \neq \emptyset & \\ \text{ in } \text{upd} \nabla' \alpha (s 4 \alpha^n) & \end{cases}
\]

\[
\text{Fig. 2.} \text{ Collapsed-jungle representation of terms.}
\]

In view of our semantic inference rules, collapsing a graph is highly beneficial: Rewriting a single node at $\alpha$ in a fully collapsed graph will effectively rewrite all subterms identical to the one that can be extracted from $\alpha$.

**Theorem 20.** The two implementations shown in Figs.\textsuperscript{1} and\textsuperscript{2} are realisations.

## 5 Transformation

It seems intuitively right that the standard graph machinery (Fig.\textsuperscript{1}) induces very little administrative overhead, whereas the collapsed-jungle graph machinery (Fig.\textsuperscript{2}) can be burdensome. We therefore propose a feasible compromise:

\[\text{The } (\cdots = \cdots)^n \text{ is a shorthand for } n \text{ equations.}\]
Optimise programs at compile time using collapsed jungles, but use standard graph machinery at run time. As we will see, it is possible to achieve some of the advantages of collapsed-jungle reduction by a source-to-source program transformation.

The program transformation we present here is a variant of supercompilation (Turchin [19, 18]), more specifically positive supercompilation (Sørensen, Glück and Jones [15]) modified to work on jungles instead of terms. The transformation process is divided into two phases. First, a finite model of the program is constructed w.r.t. a term. Second, a new program is extracted from the model.

5.1 Driving

Glück and Klimov [5] call a model of a program a process graph. The nodes in the graph are labelled by terms (i.e., program state), and each successor of a node represents a one-step unfolding. A branch in the process graph thus represents speculative execution of a particular term. Leaves in the process graph represent terms that are fully evaluated. For a particular program \( q \), each full path in a (possibly cyclic) process graph for \( q \) represents a set of actual executions of \( q \), such that the union of all full paths in the process graph includes all possible executions of \( q \).

To keep the process graph manageable, cycles are represented implicitly by leaves containing repetitions of previously seen terms; the graph thus simply becomes a tree.

To construct the process tree, we need to drive the program, that is, speculatively executing non-ground terms. For this purpose we present two modifications of the semantics of the language. The first simply allows variables in terms.

**Definition 21 (Deterministic unfolding, \( B \)).** Let the set of constructor terms \( B \) be given by the grammar \( b ::= x \mid cb^n \). The relation \( \longrightarrow \subseteq \mathcal{T} \times \mathcal{T} \) is defined as \( \rightarrow \) in Def.13 except that \( \mathcal{V} \) is replaced by \( B \) in the const rule.

The new relation \( \longrightarrow \) is still deterministic, but it allows each reduction step to ignore what corresponds to uninstantiated parts to the left of a redex. With the relation \( \rightarrow \) we can reduce non-ground terms as long as we do not run into redices of the form \( gxt^n \). To reduce such redices, we need to speculatively try out all possible forms of values of \( x \), according to the definition of \( g \).

**Definition 22 (Speculative unfolding).** The relation \( \iff \subseteq \mathcal{T} \times \mathcal{T} \) is defined as \( \iff \), but with the additional rule

\[
\begin{align*}
\langle \nabla \alpha_0 \rangle & \sim \alpha'_0 \quad \alpha'_0 \notin \mathcal{D}(\nabla) \quad (\beta = \text{FRESH})^m \\
\nabla' &= \text{upd} (\nabla + \{ \alpha \mapsto g\alpha_0 \alpha^n \}) \alpha'_0 \quad (c^m) \\
\langle \nabla + \{ \alpha \mapsto g\alpha_0 \alpha^n \} \rangle \alpha & \iff \langle \nabla' \alpha \rangle \\
\end{align*}
\]

The relation \( \iff \) is non-deterministic. When it encounters a stuck redex \( gxt^n \), it \textit{“produces”} a new DAG where \( gxt^n \) has been instantiated to \( g(cx^m)t^n \) for each pattern \( cx^m \) defined by \( g \). Each of these instantiated DAGs will allow
further reduction to take place, since each appropriate right-hand side of \( g \) now can be unfolded.

It is now easy to see how we can create a process tree for a program \( q \) and an initial term \( t \): First, pick some realisation \( I \) and label the root of the process tree by a DAG created from \( t \). Then, repeatedly add new leaves to the process tree by using the relation \( \xrightarrow{qI} \) to drive existing leaves.

### 5.2 Generalisations

Unfortunately, creating a process tree in the above manner hardly ever terminates, that is, the process tree will grow unboundedly. But, as shown by Sørensen & Glück [14], a sufficient condition for ensuring that the construction of process trees terminates, is to impose a well-quasi-order on the labels in the process tree.

**Definition 23 (wqo).** A well-quasi-order on a set \( S \) is a reflexive, transitive binary relation \( \leq \) such that, for any infinite sequence \( s_1 s_2 \ldots \) of elements from \( S \), there are \( i, j \in \mathbb{N} \) such that \( i < j \) and \( s_i \leq s_j \).

Hence, if we ensure that, for all nodes \( n \) in the process tree, there never exists an ancestor \( a \) of \( n \) such that \( \operatorname{label}(a) \leq \operatorname{label}(n) \), then all branches in the process tree will be finite. Since the process tree is finitely branching, the process tree will be finite (by König’s Lemma).

Sørensen & Glück [14] used the homeomorphic-embedding relation on terms to detect when termination is endangered; we will use this relation in ensuring termination, so a repetition is in order.

**Definition 24.** Let \( s \in \mathcal{E}, x, y \in \mathcal{X} \), and \( t, u \in \mathcal{T} \). The homeomorphic-embedding relation \( \leq \subseteq \mathcal{T} \times \mathcal{T} \) is the smallest relation satisfying the inference rules

\[
\begin{align*}
  x \leq y & \quad (t. \leq u.)^n \quad s t^n \leq s u^n \quad t \leq u_i 1 \leq i \leq n
\end{align*}
\]

Since the homeomorphic-embedding relation is a well-quasi-order, it can be used as an indicator for when to stop the development of the process tree, that is, when to stop driving. The question is then what to do, when we need to stop driving. As described by Turchin [19], we need to generalise one of the offending nodes in the process tree, in effect throwing away information that has been acquired during driving. The solution in [14] is to split up such nodes into several parts that can be explored separately. Generalisations thus give rise to branches in the process tree (as do the speculative unfolding).

### 5.3 Using DAGs

Since we employ DAGs instead of terms, our generalisation operation needs to split up DAGs. In particular, we want an operation that divide a DAG into two autonomous parts that can be expressed as terms, in order to “reassemble” the state when the transformed program is extracted from the process graph.
Example 25. Consider again the DAG from Ex. 6 shown to the left below.

\[
\begin{align*}
\text{data } \text{Pair } x y &= \langle x \rangle \langle y \rangle \\
h x &= g (f x) \\
g \langle u \rangle v &= \langle (add u) v \rangle \\
f x &= \langle (aux x) (fib x) \rangle
\end{align*}
\]

Splitting up this DAG into two non-trivial DAGs can be done as indicated in the middle. To the right is shown how such a split can be represented in the term world, interpreting the roots of lower half of the DAG as a tuple of terms.

Splitting up a DAG thus naturally gives rise to the notion of a root list of a DAG. The example should give enough intuition to support the following definitions.

**Definition 26 (Roots, ports, subdags, and proper splits).**

1. Every DAG \( \nabla \) is implicitly accompanied a finite root list, \( \text{roots}(\nabla) \in A^* \).
2. The ports of a DAG \( \nabla \) is the set of addresses outside the domain of \( \nabla \) that are reachable through the roots of \( \nabla \):
   \[
   \text{ports}(\nabla) \triangleq \{ \beta \in A \mid \beta \notin \mathcal{D}(\nabla) \land \exists \alpha \in \text{roots}(\nabla) : \alpha \rightarrow^* \beta \} .
   \]
3. A DAG \( \nabla' \) is a subdag of a DAG \( \nabla \), denoted \( \nabla' \leq \nabla \), if \( \nabla' \subseteq \nabla \) and
   \[
   \forall \alpha \in \mathcal{D}(\nabla) \setminus \mathcal{D}(\nabla') : (\alpha \rightarrow^* \beta \text{ implies } \beta \notin \{ \beta \mid \alpha \in \text{roots}(\nabla') \rightarrow^+ \beta \}) .
   \]
4. The pair \( \langle \nabla' \nabla'' \rangle \) is a proper split of \( \nabla \) if \( \nabla = \nabla' + \nabla'' \), \( \nabla'' \leq \nabla \), and \( \nabla' \neq \nabla = \nabla'' \).

Informally, all nodes external to a subdag can only reach nodes in the subdag through the roots of the subdag. That is, if \( \nabla' \leq \nabla \), then \( \nabla' \) can be “carved” out of \( \nabla \), such that \( \nabla = \nabla' + \nabla'' \) and \( \nabla'' \) interacts with \( \nabla' \) only through the roots of \( \nabla' \). A proper split is then a division of a DAG into two non-trivial parts. The split in Ex. 25 is proper.

We will, however, use such a split operation as a last resort. A more sophisticated generalisation can be achieved when the offending node \( n \) in the process tree has an ancestor \( a \) such that \( n \) is reducible to \( a \). Informally, \( \nabla \) is reducible to \( \nabla_1 \), if the terms in \( \nabla \) can be reconstructed by carving out a subdag \( \nabla_3 \) (of \( \nabla \)) and connecting it with \( \nabla_1 \) via a set of indirections \( \nabla_2 \).

**Definition 27 (terms, reducible).**

1. \( \text{terms}(\nabla) \triangleq [(\text{xtract } \langle \nabla \alpha \rangle) \mid \alpha \leftarrow \text{roots}(\nabla)].\)
2. \( \nabla \) is reducible to \( \nabla_1 \) by \( \nabla_2 = \{ (\alpha \mapsto \beta)^n \} \) and \( \nabla_3 \), if
   (a) \( \text{ports}(\nabla_1) = \{ \alpha^n \} \),
   (b) \( \text{roots}(\nabla_2) = [\alpha^n] \),
(c) \( \text{roots}(\nabla_3) = [\beta | \beta \leftarrow \text{ports}(\nabla_2)] \),
(d) \( \nabla_3 \leq \nabla \), and
(e) \( \text{terms}(\nabla) = \text{terms}(\nabla_1 + \nabla_2 + \nabla_3) \).
\[ \square \]

**Example 28.** Assume that the DAG \( \nabla_1 \) in the middle is an ancestor of the DAG \( \nabla \) to the left.

\[
\begin{align*}
\alpha & \langle xy \rangle = \ldots \\
\beta & x = a (c (d x)) \\
\gamma & \langle xy \rangle = \langle x y \rangle \\
\mu & d x = \langle 0 (s x) \rangle \\
\end{align*}
\]

The state of \( \nabla \) can be generalised into a function \( b \) by calling the function \( a \), representing state \( \nabla_1 \), by providing the arguments constructed by functions \( c \) and \( d \), representing the indirections \( \nabla_2 \) and subdag \( \nabla_3 \) to the right.

\[
\begin{align*}
\alpha & \langle xy \rangle = \ldots \\
\beta & x = a (c (d x)) \\
\gamma & \langle xy \rangle = \langle x y \rangle \\
\mu & d x = \langle 0 (s x) \rangle \\
\end{align*}
\]

We have now established some means of generalising DAGs (accompanied with root lists), and alluded to how code can be generated from such generalisations. To ensure termination of transformation, it is crucial that every generalisation breaks down a DAG into strictly smaller components.

**Remark 29.** For code-generation purposes, it is furthermore beneficial to strip every DAG for its outermost constructors and indirections by yet another generalisation step. For this presentation, however, such operation is not needed, and we therefore leave out the details.

The point of reducing a DAG to an ancestor is almost obvious: Only the subdag that has been carved out needs to be driven further. This property calls for a definition of process trees and finished nodes.

**Definition 30 (Process trees, labels, leaves, ancestors, finished).** A process tree \( \tau \) is a non-empty tree labelled with DAGs. For a particular node \( \nu \) in \( \tau \), the label of \( \nu \) is denoted \( \text{label}(\nu) \), and the set of ancestors is the set of proper predecessors of \( \nu \) in \( \tau \), denoted \( \text{anc}(\tau, n) \). The leaves of \( \tau \) are denoted by \( \text{leaves}(\tau) \). A node \( \nu \) in \( \tau \) is finished if one of the following holds.

1. \( \nu \notin \text{leaves}(\tau) \).
2. \( \exists \mu \in \text{anc}(\tau, \nu) : \text{label}(\nu) = \text{label}(\mu) \).
3. \( \text{terms}(\text{label}(\nu)) \in \mathcal{B}^* \).

A process tree is finished, if all nodes are finished. \[ \square \]
That is, a node $\nu$ in a process tree is finished if $\nu$ is an interior node, if $\nu$ is a repetition, or if there are no function symbols left to drive in $\nu$.

It remains to define exactly when to generalisations are needed. The following quasi-order seems to be desirable.

**Definition 31.** We say that $\nabla$ is *embedded in* $\nabla'$, denoted $\nabla \preceq \nabla'$, if both

$$\forall t \in \text{terms}(\nabla) : \exists u \in \text{terms}(\nabla') : t \leq u$$
$$\forall u \in \text{terms}(\nabla') : \exists t \in \text{terms}(\nabla) : t \leq u$$

We define embeddings $\nabla \nabla'$ as

$$\{ \alpha \in \mathcal{D}(\nabla') | \exists \beta \in \text{roots}(\nabla) : (\text{extract} \langle \nabla \beta \rangle) \leq (\text{extract} \langle \nabla' \alpha \rangle) \}$$

□

**Conjecture 32.** $\preceq$ is a well-quasi-order.

**Remark 33.** As of this writing, we have not been able establish a proof of the the above conjecture. If the conjecture is false, another suitable well-quasi-order needs to be invented. Leuschel [10] describes why well-quasi-orders are preferable over well-founded orders.

---

**input:** program $q$, term $t$, and a realisation $I$
**output:** the process tree $\tau$

let $\alpha_0 = \text{FRESH}$

let tree $\tau$ consist of a single node labelled $(\text{subst}_I \emptyset \alpha_0 t \phi^{-1})$ and roots $[\alpha_0]$

while $\tau$ is unfinished do

let $\nu$ be an unfinished node with $\nabla = \text{label}(\nu)$

if $\nabla'$ \relanc $(\tau, \nu)$ : label($\mu$) $\preceq \nabla$

then let $\alpha \in \text{roots}(\nabla)$ : $\exists \nabla' : (\nabla \alpha) \xrightarrow{q-l} \nabla'$

add children to $\nu$ with labels $[\nabla' | (\nabla \alpha) \xrightarrow{q-l} \nabla']$

else let $\mu \in \text{anc}(\tau, \nu)$ : $\nabla_1 = \text{label}(\mu)$ and $\nabla_1 \preceq \nabla$

if $\nabla$ is reducible to $\nabla_1$ by $\nabla_2$ and $\nabla_3$

then add three children to $\nu$ with labels $\nabla_1$, $\nabla_2$, and $\nabla_3$

else if $\exists (\nabla_2 \nabla_3) : (\nabla_2 \nabla_3)$ is a proper split of $\nabla$ and $\text{ports}(\nabla_2) \cap (\text{embeddings} \nabla_1 \nabla) \neq \emptyset$

then add two children to $\nu$ with labels $\nabla_2$ and $\nabla_3$

else let $(\nabla_2 \nabla_3)$ be a proper split of $\nabla_1$

replace all subtrees of $\mu$ with two children labelled $\nabla_2$ and $\nabla_3$

---

**Fig. 3.** The process-trees construction algorithm.

An algorithm for developing process trees is depicted in Fig. 3 assuming for the moment that relanc($\tau, \nu$) = anc($\tau, \nu$).
It turns out, however, that scrutinising all ancestors is too conservative: too many generalisations happen. Firstly, when a DAG $\nabla$ is speculatively unfolded to a DAG $\nabla'$ by an instantiation step, it is always the case that $\nabla \nless \nabla'$. Secondly, an instantiation step will give rise to a series of deterministic unfoldings (Turchin \cite{turchin} calls these transient reductions). It is well known from partial evaluation \cite{partial_evaluation} and deforestation \cite{deforestation} that such deterministic unfoldings are very beneficial, in that they are invariants in the program $q$.

We will therefore adapt a notion of relevant ancestors, as introduced in Sørensen & Glück \cite{sorensen1998}.

**Definition 34 (relevant ancestors).** Let $\nu$ be a node in a process tree $\tau$.

1. $\nu$ is generalised, if its the children have been added by a generalisation step.
2. $\nu$ is global, if its parent node is generalised, and/or if $\nu$’s children have been produced by an instantiation step (i.e., $\nu$ cannot not unfolded by $\mapsto$ alone).
3. $\nu$ is local, if it is neither generalised nor global (i.e., $\nu$ have can be unfolded by $\mapsto$).
4. The set of immediate local ancestors of $\nu$, $\text{locanc}(\tau, \nu)$, is the set of local nodes in the longest branch of local nodes $\mu_1 \ldots \mu_n$ in $\tau$ such that $\mu_n$ is the parent of $\nu$.
5. The set of relevant ancestors of $\nu$ in $\tau$ is defined as

$$\text{relanc}(\tau, \nu) \triangleq \begin{cases} \{ \mu \mid \mu \in \text{anc}(\tau, \nu) \land \mu \text{ is global} \} & \text{if } \nu \text{ is global} \\ \text{locanc}(\tau, \nu) & \text{if } \nu \text{ is local} \end{cases} \square$$

**Conjecture 35.** The algorithm in Fig.3 terminates for all programs.

Informally, the restriction to relevant ancestors is safe by the following reasoning. There cannot be a branch with an infinite number of consecutive local nodes, since then there would be an embedding, resulting in a generalisation, thus creating a global node. Since every node only can be generalised once, breaking it into strictly smaller pieces, the process tree stabilises (as a Cauchy sequence). The proposed algorithm is thus an instance of what Sørensen calls an abstract program transformer \cite{sorensen1998}. However, the above remains a conjecture, in the light of the missing proof of Conj.32.

**Theorem 36.** The algorithm in Fig.3 results in a program that is equivalent to the original program.

The efficiency of the transformed program depends on the particular realisation $I$ used by the unfolding rules. We have not been able to establish proofs of the efficiency of the transformed program with respect to $I$, but it seems likely that both of the realisations in Figs.1 and 2 will guarantee that the transformed program is at least as efficient as the original program.

\textsuperscript{2} To some extent, the efficiency also depends on the treatment of sharing between the outermost constructors; the produced code must carefully mimic such sharing.
Example 37. Let \( q \) be the Fibonacci Number program in Fig.1. Let \( t = \text{fib}\ x \). If collapsed jungles are chosen as the underlying reduction strategy, the algorithm in Fig.3 will produce the process tree depicted in Fig.4. A program very similar to the following can be extracted from the process tree:

\[
\begin{align*}
\text{data} & \quad \text{Nat} = 0 \mid s \ \text{Nat} \\
\text{data} & \quad \text{Pair} \ x \ y = \langle x \ y \rangle \\
a & \quad 0 = s \ 0 \\
a (s \ x) & \quad = b \ x \\
b & \quad 0 = s \ 0 \\
b (s \ y) & \quad = c \ (d \ y) \\
c \ (x \ y) & \quad = e \ x \ y \\
d & \quad 0 = \langle (s \ 0) \ (s \ 0) \rangle \\
d (s \ x) & \quad = f \ (d \ x) \\
e & \quad 0 \ y = y \\
e (s \ x) \ y & \quad = s \ (e \ x \ y) \\
f \langle x \ y \rangle & \quad = g \ x \ y \\
g & \quad 0 \ y = \langle y \ 0 \rangle \\
g (s \ x) \ y & \quad = h \ (i \ (g \ x \ y)) \\
h \langle x \ y \rangle & \quad = \langle (s \ y) \ x \rangle \\
i \langle x \ y \rangle & \quad = \langle (s \ y) \ x \rangle
\end{align*}
\]

The tuples in this program stem from multiple roots. Observe that, in comparison to the original, the transformed program avoids making an exponential number of calls.  

6 Conclusion and Related Work

The benefits of deforestation and supercompilation are well illustrated in the literature, and advances in ensuring termination of these (and similar) transformations have greatly improved their potential as automatic, off-the-shelf optimisation techniques. One problem, however, remains in making these techniques suitable for inclusion in the standard tool-box employed by compiler writers: It is in general not possible to ensure that a transformed program is at least as efficient as the original program, without imposing severe (usually syntactic) restrictions on the original programs.

In this paper we have tried to formulate a version of positive supercompilation that addresses the concern of ensuring efficiency of the transformed program. The key ingredient in this formulation is the return to viewing terms as graphs.

In the first part of this paper, we have shown that, for a small function-oriented programming language, any graph-reduction implementation obeying two reasonable rules will lead to the same semantics. We have given two examples of such implementations, one similar to call-by-need, and one similar to collapsed-jungle evaluation.

Wadsworth \[22\] invented call-by-need for the pure \( \lambda \)-calculus, and proved that normal-order (call-by-need) graph reduction is at least as efficient as normal-order term reduction for a certain subset of graphs representing \( \lambda \)-terms, and he devised an algorithm for performing normal-order reduction. Hoffmann & Plump \[7\], the main source of inspiration for this research, have proved that term rewrite systems could be translated into hypergraph replacement systems. They define the notion of fully-collapsed jungles in terms of morphisms on graphs, and they show uniqueness of such fully-collapsed graphs. The collapse-function given in our second realisation of graph reduction is basically an implementation of
their fold-morphism. Their main focus, however, is on showing that confluence and termination is preserved for a large class of term rewrite systems.

In the second part of this paper, we have presented a version of supercompilation that — when using collapsed jungles as the underlying representation — can give some of the speedups that collapsed-jungle evaluation can give, but without any run-time overhead. In this respect, we have effectively achieved to perform tupling (Pettorossi [12] and Chin [2]), an aggressive, semi-automatic program transformation based on the unfold/fold framework (Burstall & Darlington [1]). The key ingredient in tupling is to discover a set of progressive cuts [12] in the call graph for a program, and automatic search procedures for such cuts have been investigated intensively. In particular, Pettorossi, Pietropoli & Proietti [11] manipulate DAGs in a fashion that is very similar to our notion of a proper split. It seems that we are able to synchronise common calls, because we use of a local/global unfolding strategy similar to what is used in partial deduction (see e.g., Gallagher [3] or De Schreye, et.al. [3]).

Further work needs to be done in three directions. Firstly, we need to prove the efficiency and correctness properties conjectured in this paper. Secondly, we want to investigate the exact relationship between tupling and graph-based supercompilation. Thirdly, to establish empirical results, an implementation of the presented transformer is under construction. In the future, we hope to bootstrap the transformer, in the sense of expressing it in terms of the subject language. Having done this, it will be possible to experiment with self-application (e.g., as described by Jones, Sestoft, and Søndergaard [8] or Turchin [20]).

Acknowledgements

I want to thank Robert Glück, Neil D. Jones, Michael Leuschel, Oege de Moor, Henning Niss, and (certainly not least) Morten Heine Sørensen for enlightening discussions. I also want to thank the anonymous referees for their detailed comments and suggestions for improvements.

References


3 Maurizio Proietti kindly provided this paper. Unfortunately it arrived very close to the deadline, so the precise relationship is unclear as of this writing.
Fig. 4. Process tree of the Fibonacci program. Global nodes are shaded.


Higher-Order Pattern Matching for Automatically Applying Fusion Transformations

Ganesh Sittampalam and Oege de Moor
Programming Research Group, Oxford University Computing Laboratory, Wolfson Building, Parks Road, OX1 3QD, United Kingdom
{ganesh,oege}@comlab.ox.ac.uk

Abstract. We give an algorithm for higher-order pattern matching in the context of automatic program transformation. In particular, we show how accumulating parameter optimisations of functional programs can be automatically derived with the aid of programmer annotations. These techniques have been successfully applied to some complex manual derivations in the literature, such as Bird’s “longest path-sequence”.

1 Background and Motivation

Consider the following program for calculating the minimum depth of a leaf-labelled binary tree. The language we use is Haskell [3] but it could easily be rewritten in another functional language.

```
data Tree α = Leaf α | Bin (Tree α) (Tree α)
mindepth (Leaf x) = 0
mindepth (Bin s t) = 1 + min (mindepth s) (mindepth t)
```

The program is short, clearly stated, and it is easy to see that it will do what we intend. However, it is also not very efficient. Imagine a right-leaning tree; our program will quickly explore the left branch of the tree, but will then spend a significant amount of time in the right-hand branch despite the fact that it will quite soon become apparent that the leaf of minimum depth must be in the left branch and that the search of the right branch could be aborted.

A little bit of thought allows an experienced programmer to make this definition more efficient; simply add two accumulating parameters to keep track of the minimum depth found so far in the tree and the current depth, and then the search can be cut off for any particular branch if the current depth reaches the minimum already found.

It should be noted that this the following is still not the fastest possible program in all situations, since it still carries out a depth-first search, but it does represent a significant improvement over the original program; this optimisation is also a simple representative of a larger class of optimisations on tree-consuming functions, such as the alpha-beta algorithm for fast searches of game trees [4].

© Springer-Verlag Berlin Heidelberg 2001
\[
\begin{align*}
md (\text{Leaf } x) d m &= \min d m \\
md (\text{Bin } s t) d m &= \text{if } nd \geq m \text{ then } m \\
&\quad \text{else } \text{md } s \text{ nd } (\text{md } t \text{ nd } m) \\
\text{where } nd &= d + 1
\end{align*}
\]

\[
\text{mindepth } t = \text{md } t \ 0 \ \infty
\]

However, it is not nearly so obvious that the resulting program is correct, and it is certainly much harder to understand than the above code, which increases the likelihood of bugs and makes it harder to maintain. What we need is a programming paradigm that combines the best of both worlds – the clarity of the first program and the efficiency of the second.

The key point to note here is that the first and second programs can be related by equational reasoning, and that in fact once we have made the insight that we should use “minimum depth found so far” and “current depth” as accumulating parameters, deriving the second program from the first is quite straightforward.

Thus, it seems logical to take the approach of giving our program as the first piece of code, together with information on how to derive the more efficient second program. We will end up with source that is much easier to read and maintain, and with efficient object code that we can be confident is correct, assuming we trust our compiler. In fact, we can take this idea a stage further and emit the second program as an intermediate result, thus enabling the programmer to have confidence that the optimisation he or she intended has actually been applied. It is the development of an algorithm that will make such an approach possible for programs like \text{mindepth} that we will discuss in this paper.

1.1 Fusion Transformations

One feature of the \text{mindepth} program is that it is a recursive traversal of the inductive datatype \text{Tree}. This means that it can be expressed as an instance of a general function known as a \text{fold}, which encapsulates this pattern of programming. A separate fold function exists for each datatype; in this instance it is defined as follows:

\[
\begin{align*}
\text{foldbtree } n \ l \ (\text{Leaf } a) &= l \ a \\
\text{foldbtree } n \ l \ (\text{Node } s \ t) &= n \ (\text{foldbtree } n \ l \ s) \ (\text{foldbtree } n \ l \ t)
\end{align*}
\]

The advantage of expressing our program as a fold is that it allows us to make use of a general rule about folds known as \text{fusion} \[2\]. This rule essentially states that under certain conditions we can replace a fold followed by the application of another function with just a single fold.

In the case of the example above we can specify the \text{md} function from the second program in terms of the original \text{mindepth} function we gave by the definition

\[
\text{md } t \ d \ m = \min (\text{mindepth } t + d) m
\]
This specification shows us that \textit{md} can be expressed as a function applied to the result of a fold (the fold being \textit{mindepth}), and it turns out that the fusion rule applies and in fact mirrors the manual derivation required to get from the first program to the second program. Since it tends to be difficult to direct the progress of automatic derivations, the use of this rule as a general "template" allows us to concisely specify the path a transformation system should take. Once we have done the appropriate transformation on \textit{md}, we can redefine \textit{mindepth} by

\[
\text{mindepth } t = \text{md } t \ 0 \ \infty
\]

For each datatype, a different fusion rule exists to go with the fold function; given the datatype, it is reasonably easy to derive the appropriate fusion rule automatically. In the case of the leaf-labelled binary trees described above, the appropriate rule is:

\[
f(\text{foldbtree } n \ l \ t) = \text{foldbtree } n' \ l' \ t \\
\text{if } \forall a. l' a = f(l \ a) \\
\forall b c. n' (f \ b) (f \ c) = f(n \ b \ c)
\]

Applying this rule automatically is an application of \textit{term rewriting}; a process of applying equational rules such as the one above to a program. The first step is to look for instances of the left-hand side of the rule in the program we are manipulating. Once we find one, we know that we can replace it with the right-hand side if we can satisfy the side conditions given.

We take the conditions one by one; first we transform both sides of each condition by repeatedly applying term rewriting. In practice the rules that are used here will turn out to be simple ones with no side conditions, so this is quite a straightforward process. Once no more rules apply, our next problem is to find instantiations for the unknowns that will make the condition true. It will always be the case that unknowns will only occur on the left-hand side of each condition, which makes this task somewhat simpler; however they are of function type, which means that we are forced to actually synthesise new function definitions.

This procedure is best illustrated with an example. We have developed a system MAG \cite{MAG} which carries out derivations of this nature, and the actual calculation for \textit{mindepth} it carries out can be found in the appendix. The first point to note is that we use a trick of "seeding" the fusion to avoid the programmer having to give their original program in terms of a fold. Next, we examine in detail the application of the \textit{treefusion} rule. This contains two sub-calculations that show the exhaustive rewriting of the right-hand sides of the above side conditions (the universal quantification is expressed as equality of \(\lambda\)-abstractions). The final result of the second of these calculations is
\[ \lambda b c d e. \text{if } a >= e \text{ then } e \text{ else } \min (\mindepth b + a) (\min (\mindepth c + a) e) \]

\text{where } a = 1 + d

Now, instantiating \( f \) in \( \lambda bc.n'(fb)(fc) \), which is the left-hand side of the side condition in question, gives

\[ \lambda b c . n' (\lambda h i . \min (\mindepth b + h) i)(\lambda j k . \min (\mindepth c + j) k) \]

Thus, the substitution we want for \( n' \) is as follows; we will have to synthesise a rather complicated function body.

\[ n' := \lambda fg d e . \text{if } a >= e \text{ then } e \text{ else } f a (g a e) \]

\text{where } a = 1 + d

This paper is concerned with an algorithm for doing this. In previous work \[8\] we have given one such algorithm, known as the “one-step” algorithm; this algorithm represented an advance on the standard algorithm in the literature which was developed by Huet and Lang \[10\]; for example it allowed certain programming examples such as the well known “fast reverse” optimisation to be derived. However, this algorithm proves to be inadequate for more complicated problems such as this one, and therefore we now extend the ideas presented there to produce the (predictably named) \textit{two-step} algorithm, which has rather more limited applicability but turns out to be able to find the matches we need in those cases where the one-step algorithm is inadequate.

Of course, we are not the only people to have worked on automatically applying this and similar sorts of transformation. For example Chitil \[5\] shows how shortcut deforestation \[9\] can be carried out based on type inference, and Onoue et al \[14\] have implemented similar transformations in an optimising pass to ghc (the Glasgow Haskell Compiler); ghc also implements short-cut deforestation via programmer-specified rules.

\subsection*{1.2 Higher-Order Matching}

Abstracting from the particular programming language being used, we state the problem as follows. Given \( \lambda \)-expressions \( P \) (the \textit{pattern}) and \( T \) (the \textit{term}), find a substitution \( \phi \) such that

\[ \phi P = T \]

Here equality is taken modulo renaming (\( \alpha \)-conversion), elimination of redundant abstractions (\( \eta \)-conversion), and substitution of arguments for parameters (\( \beta \)-conversion). A substitution \( \phi \) that satisfies the above equation is said to be a \textit{match}. Later on, we shall refine our definition of equality of \( \lambda \)-expressions and the notion of a match.

There is no canonical choice for \( \phi \). For example, let

\[ P = fx \text{ and } T = 0 \]
Possible choices for $\phi$ include:

$$f := (\lambda a.a) \text{ and } x := 0,$$
$$f := (\lambda a.0),$$
$$f := (\lambda g.g \ 0) \text{ and } x := (\lambda a.a),$$
$$f := (\lambda g.g \ (g \ 0)) \text{ and } x := (\lambda a.a),$$
$$\ldots$$

All these matches are incomparable in the sense that they are not substitution instances of each other.

It should be noted that matching differs from the more commonly known problem of unification in that free variables are allowed in only one of the $\lambda$-expressions; this is an acceptable restriction because the side conditions we will need to satisfy will only have free variables on the left-hand side.

2 Preliminaries and Specification

We start by introducing some notation, and then pin down the matching problem that we intend to solve. Users of our algorithm (for instance those who wish to understand the operation of MAG) need to know only about this section of the paper.

2.1 Expressions

An expression is a constant, a variable, a $\lambda$-abstraction or an application. There are two types of variables: bound (local) variables and free (pattern) variables. We shall write $a, b, c$ for constants, $x, y, z$ for local variables, $p, q, r$ for pattern variables, and use capital identifiers for expressions. Furthermore, function applications are written $FE$, and lambda abstractions are written $\lambda x.E$. As usual, application associates to the left, so that $E_1 E_2 E_3 = (E_1 E_2) E_3$.

It is admittedly unattractive to make a notational distinction between local and pattern variables, but the alternatives (De Bruijn numbering or explicit environments) would unduly clutter the presentation. In the same vein, we shall ignore all problems involving renaming and variable capture, implicitly assuming that identifiers are chosen to be fresh, or that they are renamed as needed. Equality ($=$) is modulo renaming of bound variables ($\alpha$-conversion).

Besides renaming, we also consider equality modulo the elimination of superfluous arguments. The $\eta$-conversion rule states that $(\lambda x.E \ x)$ can be written as $E$, provided $x$ is not free in $E$. An expression of this form is known as an $\eta$-redex; we shall write $E_1 \simeq E_2$ to indicate that $E_1$ and $E_2$ can be converted into each other by repeated application of $\eta$-conversion and renaming.

The $\beta$-conversion rule states how arguments are substituted for parameters: $(\lambda x.E_1) \ E_2$ is converted to $(x := E_2)E_1$. A subexpression of this form is known as a $\beta$-redex. The application of this rule in a left-to-right direction is known
as $\beta$-reduction. Unlike $\eta$-reduction, repeated application of $\beta$-reduction is not guaranteed to terminate.

An expression is said to be normal if it does not contain any $\eta$-redex or $\beta$-redex as a subexpression. An expression is closed if all the variables it contains are bound by an enclosing $\lambda$-abstraction.

Some readers may find it surprising that we have chosen to work with un-typed $\lambda$-expressions, instead of committing ourselves to a particular type system. Our response is that types could be represented explicitly in expressions (as in Girard’s second-order $\lambda$-calculus, which forms the core language of ghc [15]). Our algorithm can be adapted accordingly to expressions in which types are explicit in the syntax. However, as with the unification algorithm presented in [12], it does not depend on a particular typing discipline for its correctness.

2.2 Substitutions

A substitution is a total function mapping pattern variables to expressions. Substitutions are denoted by Greek identifiers. We shall often specify a substitution by listing those assignments to variables that are not the identity. Substitutions are applied to expressions in the obvious manner. Composition of substitutions $\phi \circ \psi$ is defined by first applying $\psi$ and then $\phi$.

We say that one substitution $\phi$ is more general than another substitution $\psi$ if there exists a third substitution $\delta$ such that $\psi = \delta \circ \phi$; we also write $\phi \leq \psi$. Intuitively, when $\phi \leq \psi$, the larger substitution $\psi$ substitutes for variables that $\phi$ leaves alone, or it makes more specific substitutions for the same variables.

A substitution is said to be normal if all expressions in its range are normal, and closed if any variables that it changes are mapped to closed expressions.

2.3 Rules

A rule is a pair of expressions, written $(P \rightarrow T)$, where $P$ does not contain any $\eta$-redexes, and $T$ is normal, with all variables in $T$ being local variables, i.e. they occur under an enclosing $\lambda$-abstraction. The matching process starts off with $T$ closed, but because it proceeds by structural recursion it can generate new rules which do not have $T$ closed. In such a rule, a variable is still regarded as being local if it occurred under an enclosing $\lambda$-abstraction in the original rule. We call $P$ the pattern and $T$ the term of the rule. Rules are denoted by variables $X$, $Y$ and $Z$; sets of rules are denoted by $Xs$, $Ys$ and $Zs$.

The pattern $P$ is also restricted as follows. For each subexpression $FE$ of $P$ such that $F$ is flexible, i.e. has a free variable as its head, $E$ must satisfy the following conditions. Firstly it must contain no pattern variables. Secondly, suppose $E = \lambda x_1...x_n.B$, where $n$ is possibly 0, but $B$ does not contain any outermost $\lambda$s. Then each of $x_1...x_n$ must occur at least once in $B$, and $B$ must contain at least one constant symbol, or alternatively a local variable that is bound in $P$ outside $FE$. So for example, $p(\lambda x.x + x)$ is a valid pattern, but $p(\lambda x.x)$ is not, and neither is $p(\lambda x.0)$.
This last restriction in particular will seem rather arbitrary, but it is this that will allow us to produce a workable algorithm for solving rules. In cases where this restriction prevents the algorithm from being used, the one-step matching algorithm that we have previously developed [8] can be applied instead, and so it is not in fact a bar to carrying out the derivations we are interested in, such as that for mindepth. Others [12, 13] have introduced even more severe restrictions on patterns for the purposes of developing useful matching or unification algorithms.

The application of a substitution to a rule is defined by $\sigma(P \rightarrow T) = \sigma P \rightarrow T$ (since $T$ is closed there is no point in applying a substitution to it). The obvious extension of this definition to a set of rules applies.

A substitution $\phi$ is said to be pertinent to a rule $(P \rightarrow T)$ if all variables it changes are contained in $P$. Similarly, a substitution is pertinent to a set of rules if all variables it changes are contained in the pattern of one of the rules.

### 2.4 Restricting $\beta$-Reduction

Thus far, we have established a definition of equality ($\simeq$) that takes account of $\alpha$- and $\eta$-conversion, but gives no regard to $\beta$-reduction. The reason for this is that $\beta$-reduction makes quite complex equivalences possible; for example the expression $(\lambda xy.x + y)((\lambda z.z)0)0$ is $\beta$-equivalent to $0 + 0$.

Given a rule $(P \rightarrow T)$, we know that $T$ is normal. Thus, if we are looking for a substitution $\phi$ that makes $P$ and $T$ equivalent, it would make sense to specify this by defining a function betanormalise that exhaustively applies $\beta$-reduction, and then stating that we want all $\phi$ such that betanormalise ($\phi P$) $\simeq$ $T$.

Unfortunately, finding all such substitutions $\phi$ is a hard problem; in fact it is not even known whether it is possible to decide whether any exist or not. Therefore, our approach is to choose a restriction to this specification that makes the problem tractable but still produces the results we need for our particular application.

The form of the specification above also gives us a clue as to how to go about restricting it. One way we could write the betanormalise function described above would be:

\begin{align*}
\text{betanormalise } c &= c \\
\text{betanormalise } x &= x \\
\text{betanormalise } p &= p \\
\text{betanormalise } (\lambda x . E) &= \lambda x . \text{betanormalise } E \\
\text{betanormalise } (E_1 E_2) &= \text{case } E'_1 \text{ of} \\
&\quad (\lambda x . B) \rightarrow \text{betanormalise } ((x := E'_2) B) \\
&\quad \rightarrow (E'_1 E'_2) \\
\text{where } E'_1 &= \text{betanormalise } E_1 \\
E'_2 &= \text{betanormalise } E_2
\end{align*}

Essentially betanormalise conducts a tree walk, reducing $\beta$-redexes as it finds them. The key point to note is that because reducing one can cause others to
become visible, it is necessary to recursively call betanormalise on the result of doing this. When trying to develop a matching algorithm, we find that it is this fact in particular that makes the problem very hard. Thus, we are led to the following definition (we omit the obvious cases that simply recurse through the tree in the same way as betanormalise):

\[
\text{twostep}(E_1, E_2) = \text{case } E'_1 \text{ of } \begin{align*}
(\lambda x . B) & \rightarrow \text{unmark} \left(\text{markedstep} \left(\left(x := \text{mark} E'_2\right) B\right)\right) \\
- & \rightarrow (E'_1, E'_2) \\
\text{where} & \quad E'_1 = \text{twostep} E_1 \\
& \quad E'_2 = \text{twostep} E_2
\end{align*}
\]

Here we have written a function with a similar definition to betanormalise, but we have replaced the recursive call that occurs after reducing a \(\beta\)-redex with something slightly different; the idea is that we will make precisely two passes over any particular \(\beta\)-redex. Thus, the combination of the functions \text{mark} and \text{markedstep} is designed to apply a substitution and only reduce redexes that result from the application of the substitution. We use \text{mark} to label all the outer lambda abstractions in an expression:

\[
\text{mark} (\lambda x . E) = \lambda x' . (\text{mark} E) \\
\text{mark} E = E
\]

The function \text{markedstep} is another function that follows the same pattern as betanormalise; in this case it only reduces redexes whose left-hand side is a marked lambda, and then does nothing more to the result. Finally the function \text{unmark} removes any remaining marks; its definition is obvious. As before we leave out the obvious cases from the definition of \text{markedstep}'.

\[
\text{markedstep}(E_1, E_2) = \text{case } E'_1 \text{ of } \begin{align*}
(\lambda' x . B) & \rightarrow (x := E'_2) B \\
- & \rightarrow (E'_1, E'_2) \\
\text{where} & \quad E'_1 = \text{markedstep} E_1 \\
& \quad E'_2 = \text{markedstep} E_2
\end{align*}
\]

What we are aiming at with this (admittedly rather complicated) function \text{twostep} is something that will reduce a certain class of expressions to their normal form. Very informally, the expressions we want this to be true for are those which contain subexpressions that are lambda abstractions applied to other lambda abstractions, \textit{i.e.} those of similar form to \((\lambda y.y 0)(\lambda x.x + x)\).

Of course this class will also contain simpler expressions, for example those with subexpressions of the form \((\lambda x.x + x)\). However, it does \textit{not} contain those of the form \((\lambda z. (\lambda x.x + x))(\lambda y.y 0)\); that is those which would require three (or more) passes over the \(\beta\)-redex to reduce it to normal form.

Readers familiar with the theory of the \(\lambda\)-calculus will recognise a strong similarity between our definition and that of \textit{finite developments} \cite{1}, which partially reduce a term by first underlining all \(\beta\)-redexes in the original term and then
reducing all underlined redexes. However, *twostep* is subtly different; consider it as a two-pass function where β-redexes are first reduced and the results are then passed to *markedstep* for the second pass. Then whereas a single complete finite development would only reduce the outermost redex in \((\lambda xyz.x + y + z)\)1 2 3, *twostep* would reduce this expression completely in the first pass.

The more common way to restrict the matching problem is to limit the order of the terms in the images of the substitutions returned; a *first-order* term is a ground term such as 0, a *second-order* term is a function which takes first-order terms as parameters, and so on. However, this approach does not sit well with our particular application; the standard Huet and Lang algorithm [10] returns only second-order matches which are not enough to solve problems such as mindepth, and although a general algorithm for finding third-order matches exists [6], the set of these can be infinite, and a matching algorithm that is not guaranteed to terminate is not useful in a practical program transformation system.

Another restriction that has been explored in [12, 13] is that of higher-order patterns. These impose a rather more stringent restriction than ours, namely that any free variable must only have distinct bound variables as arguments. This restriction makes unification (and thus matching) decidable, and since terms of this form appear frequently in theorem-proving situations it is an appropriate restriction to impose in such environments. However, the matching problems generated by the transformations we are interested in do not satisfy this restriction.

2.5 Two-Step Matches

Having carefully designed our restriction of β-reduction, we can now specify exactly what results our algorithm will produce.

A rule \((P \to T)\) is *satisfied* by a normal substitution \(\phi\) if

\[
\text{twostep}(\phi P) \simeq T.
\]

The substitution \(\phi\) is then said to be a *two-step match*. Note that we take equality not only modulo renaming, but also modulo η-conversion. A normal substitution satisfies a set of rules if it satisfies all elements of that set. We write \(\phi \vdash X\) to indicate that \(\phi\) satisfies a rule \(X\), and also \(\phi \vdash Xs\) to indicate that \(\phi\) satisfies a set of rules \(Xs\).

The notion of a two-step match contrasts with that of a general match because of our restriction of the notion of equality; a normal substitution \(\phi\) is said to be a general match if betanormalise \((\phi P) \simeq T\). In [8], we defined one-step matches to be those \(\phi\) that satisfy step \((\phi P) \simeq T\), where step is defined (in a similar way to twostep etc above) by

\[
\text{step}\ (E_1 E_2) = \text{case } E'_1 \text{ of }
\]

\[
(\lambda x . B) \to (x := E'_2) B
\]

\[
- \to (E'_1 \ E'_2)
\]

where \(E'_1 = \text{step } E_1\)

\(E'_2 = \text{step } E_2\)

For convenience we shall refer to a two-step match simply as a *match*. 
2.6 Match Sets

Let $Xs$ be a set of rules. A match set of $Xs$ is a set $M$ of normal substitutions such that:

- For all normal $\phi$: $\phi \vdash Xs$ if and only if there exists $\psi \in M$ such that $\psi \leq \phi$.
- For all $\phi_1, \phi_2 \in M$: if $\phi_1 \leq \phi_2$, then $\phi_1 = \phi_2$.

The first condition is a soundness and completeness property. The backwards direction is soundness; it says that all substitutions in a match set satisfy the rules. The forwards implication is completeness; it says that every match is represented. The second condition states that there are no redundant elements in a match set.

For example, if $Xs = \{p (\lambda x.x + x) \to 0 + 0\}$, then

$$\{\{p := (\lambda y.0 + 0)\}, \{p := (\lambda y.0)\}\}$$

is a match set. But if $Xs = \{p (\lambda y.y0) \to 0 + 0\}$, then since

$$\text{betanormalise} (\lambda z.z (\lambda x.x + x) (\lambda y.y0)) = 0 + 0$$

we have that

$$\{p := (\lambda z. (\lambda x.x + x))\}$$

is a general match, but because

$$\text{twostep} (\lambda z.z (\lambda x.x + x) (\lambda y.y0)) = (\lambda x.x + x) 0$$

it is not a member of the match set.

In general, match sets are unique up to pattern variable renaming, and consequently we shall speak of the match set of a set of rules. In the remainder of this paper, we present an algorithm that computes match sets; we shall omit the proof that this algorithm is correct, but will sketch a proof that that match sets will include all third-order general matches.

First, we show that twostep is equivalent to full $\beta$-reduction for all terms of third-order or below. Then, since general matches satisfy the specification $\text{betanormalise} (\phi P) \simeq T$, all third-order general matches must also satisfy $\text{twostep} (\phi P) \simeq T$ and will thus be two-step matches as well.

Showing the equivalence of twostep and $\beta$-reduction involves tracking $\beta$-redexes from the parameter to the result of twostep. Suppose that in the redex $(\lambda x.B) T$, $\lambda x.B$ is of order $n$. Then $T$ must be of order at most $n - 1$, and thus any new redexes created by reducing $(\lambda x.B) T$ to $(x := T) B$ must be of order at most $n - 1$ too.

Now, if $E$ is a term, then in calculating twostep $E$ any $\beta$-redex will be reduced once and then any resulting redexes will be reduced again by markedstep. Thus, if a redex of order $n$ exists in twostep $E$, one of order $n + 2$ or higher must exist in $E$. Since first-order $\beta$-redexes are not possible, it must be the case that all redexes of third-order or below are completely removed by twostep, and hence it is equivalent to full $\beta$-reduction for terms of third-order or below. This completes our proof.
3 Outline of an Algorithm

Our matching algorithm operates by progressively breaking down a set of rules until there are none left to solve. We outline its structure, then give the details of the function that will implement the key part of our algorithm.

The function *matches* takes a set of rules and returns a match set. It is defined recursively (using the notation of Haskell [3]):

\[
\text{matches} :: \text{[Rule]} \rightarrow \text{[Subst]}
\]

\[
\text{matches} \ [\] = \text{[idSubst]}
\]

\[
\text{matches} \ (X : \text{Xs}) = \text{[(φ ◦ σ) | (σ, Ys) ← resolveX, φ ← matches (σ (Ys + Xs))]}\]

That is, the empty set of rules has the singleton set containing the identity substitution as a match set. For a non-empty set of rules \((X : \text{Xs})\), we take the first rule \(X\) and break it down into a (possibly empty) set of smaller rules \(Ys\) together with a substitution \(σ\) which makes \(Ys\) equivalent to \(X\). We then combine the \(Ys\) with \(Xs\), the remainder of the original rules, apply \(σ\), and return the results of a recursive call to *matches* combined with \(σ\).

Clearly it would be advantageous to arrange the rules in such a manner that we first consider rules where *resolveX* is small, perhaps only a singleton. There is no particular reason why we should take the union of \(Ys\) and \(Xs\) by list concatenation: we could place ‘cheap’ rules at the front, and ‘expensive’ rules at the back.

The function that breaks up \(X\) into smaller rules is called *resolve*. Readers who are familiar with the logic programming paradigm will recognise it as being analogous to the concept of “resolution”. We specify the behaviour of *resolve* through certain properties; let

\[
[(σ_0, Ys_0), (σ_1, Ys_1), \ldots, (σ_k, Ys_k)] = \text{resolveX}.
\]

We require that

- For all normal substitutions \(φ\):
  \[
  (φ \vdash X) \equiv \bigvee_i (φ \vdash Ys_i \land σ_i ≤ φ).
  \]

- For all normal substitutions \(φ\) and indices \(i\) and \(j\):
  \[
  (φ \vdash Ys_i) \land (φ \vdash Ys_j) \Rightarrow i = j.
  \]

- For each index \(i\), \(σ_i\) is pertinent to \(X\), closed and normal.
- The pattern variables in \(Ys_i\) are contained in the pattern variables of \(X\).
- For each index \(i\):
  \[
  Ys_i \ll X.
  \]
The first of these is a soundness and completeness condition: it says that all relevant matches can be reached via \texttt{resolve}, and that \texttt{resolve} stays true to the original set of rules. The second condition states that \texttt{resolve} should not return any superfluous results. The third and fourth conditions are technical requirements we need to prove the non-redundancy of \textit{matches}. Finally, the last condition states that we make progress by applying \texttt{resolve}; \textit{i.e.} that the process of breaking down the set of rules will eventually terminate.

### 3.1 Defining \texttt{Resolve}

Defining \texttt{resolve} is done on a case-by-case basis depending on the syntactic structure of the argument rule; the individual cases are summarised in the table below, the intention being that the first applicable clause is used. The reader is reminded of the notational distinction we make between variables: \(x\) and \(y\) represent local variables, \(a\) and \(b\) constants, and \(p\) a pattern variable.

<table>
<thead>
<tr>
<th>(X)</th>
<th>(\text{resolve } X)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x \rightarrow y)</td>
<td>([(id, [])], \text{ if } x = y)</td>
</tr>
<tr>
<td>(\lambda x.P \rightarrow \lambda x.T)</td>
<td>([(id, [P \rightarrow T])]])</td>
</tr>
<tr>
<td>((F E) \rightarrow T, F \text{ flexible})</td>
<td>([(id, [F \rightarrow \text{etaRed}(\lambda x.B)])], \text{ if } x\ \text{fresh, } B \leftarrow \text{abstracts } x ET])</td>
</tr>
<tr>
<td>(P \rightarrow T)</td>
<td>([], \text{ otherwise})</td>
</tr>
</tbody>
</table>

Let us now examine each of these clauses in turn.

The first clause says that two local variables match only if they are equal.

The second clause says that we can solve a rule \((p \rightarrow T)\) where the pattern is a pattern variable by making an appropriate substitution. Such a substitution can only be made, however, if \(T\) does not contain any local variables occurring without their enclosing \(\lambda\): since the original term cannot contain any pattern variables, any variables in \(T\) must have been bound in the original term and so the substitution would move these variables out of scope.

The third clause deals with matching of constants \(a\) and \(b\). These only match when they are equal.

Next, we consider matching of \(\lambda\)-abstractions \((\lambda x.P)\) and \((\lambda x.T)\). Here it is assumed that the clauses are applied modulo renaming, so that the bound variable on both sides is the same, namely \(x\). To match the \(\lambda\)-abstractions is to match their bodies.
Recall, however, that we took equality in the definition of matching not only modulo renaming, but also modulo $\eta$-conversion. We therefore have to cater for the possibility that the pattern contains a $\lambda$-abstraction, but the term (which was assumed to be normal) does not. This is the purpose of the clause for matching $(\lambda x.P)$ against a term $T$ that is not an abstraction: we simply expand $T$ to $(\lambda x.T\,x)$ and then proceed as with the previous clause.

The next two clauses deal with the cases where the pattern is an application. Given a rule $FE \to T$, the aim is to break it down into new rules by finding pairs of terms against which to match $F$ and $E$ respectively. The second of the two clauses is simple; it deals with the case where the function part of the pattern is not flexible, i.e. it does not have a free variable at its head and therefore there is no point in trying to construct a new function to match against it; if the term is also an application we simply match up the functions and arguments respectively, and if not we do nothing.

The difficult case arises when the function part of the pattern is flexible; it is at this stage that the novel part of our algorithm comes into play, and our treatment of this case is described in detail in the following section. In essence, the $abstracts$ function uses structure of $E$ and $T$ to “guess” at possible bodies for a function against which $F$ could be matched; an enclosing $\lambda$ is then added, and if a top-level $\eta$-redex exists it is stripped off by the function $\etaRed$.

The final rule states that if none of the above rules were applicable then no matches exist.

### 3.2 Abstracting New Function Bodies

The $abstracts$ function is used when resolving a rule of the form $(FE \to T)$, where $F$ is flexible and $E$ satisfies the restrictions described in section 2.3. As we remarked earlier, the overall goal when breaking down this rule is to generate new rules by finding pairs of terms against which to match $F$ and $E$ respectively. Since $E$ does not contain pattern variables, the only term it can match is $E$ itself, and thus the problem is to find valid terms to match $F$ against.

In other words, given the rule $(FE \to T)$, we would like to generate a new rule $(F \to T')$ which is satisfied by precisely the same match set.

Now, consider the $twostep$ function. Suppose that $\phi \vdash (FE \to T)$; then $twostep(\phi(E)) \simeq T$, and so $twostep((\phi(F))E) \simeq T$, since $E$ has no pattern variables. Next, assume that $twostep(\phi(F))$ is a $\lambda$-abstraction, of the form $\lambda x.B$; although this is a technically incorrect simplification, it turns out that this is correctly balanced by the use of $\etaRed$ described above.

Then the definition of $twostep$ tells us that $unmark(markedstep((x := mark\,E)\,B)) = T$; thus for each $B$ that satisfies this equation, $\phi \vdash (FE \to T)$ will imply $\phi \vdash F \to \lambda x.B$; some rather more rigorous reasoning than the above shows that if we find all such $B$, the appropriate reverse implication will hold, satisfying the completeness of $resolve$.

Thus, the goal of the function $abstracts$ is to find values for $B$. It takes as parameters a variable $x$, the expression $E$ and the term $T$, which have the roles as outlined above. Clearly one possible value for $B$ is $T$ itself; but consider the
changes unmark (markedstep ((x := mark E) B)) makes to the expression B: it replaces all occurrences of x in B with E and then performs one β-reduction pass over the newly created occurrences of E and their arguments. Thus, when finding values for B, we should look for subexpressions in T that are the result of performing a β-reduction pass over E applied to a set of arguments; we can then (selectively) replace these subexpressions with the variable x applied to this set of arguments. We call such subexpressions instances of E in T, and the process of replacing them with the variable x applied to appropriate arguments is known as abstracting.

For example, if T is 1 + (0 + 0) and E is \( \lambda y.y + y \), then the subexpression 0 + 0 is the only instance of E in T, and can be replaced with x 0. Thus the result of abstracts \( x ET \) will be \{1 + (0 + 0), 1 + x 0\}.

This procedure is somewhat complicated by the fact that instances may overlap; for example consider T = (1 + 1) + (1 + 1) and E = \( \lambda y.y + y \); then both the entire term T and the two occurrences of (1 + 1) are instances of E in T. It is for this reason that our algorithm searches for instances by an iterative process, which we shall now describe.

The function abstracts is the union of the results of applying the function abstracts\(_n\) to the same arguments for all \( n \). The contents of abstracts\(_n\) \( x ET \) will be all the possible ways in which precisely \( n \) instances of E in T can be abstracted.

\[
abstracts x ET = \bigcup \{ \text{abstracts}_n x ET \mid n = 0\ldots \}
\]

Clearly, abstracting 0 instances will just give us T, so abstracts\(_0\) just returns \{T\}. If we know all expressions in which \( n \) instances have been abstracted, then we can generate all those in which \( n + 1 \) instances by simply abstracting one more instance in all possible ways from each. Since the body of E must contain at least one constant or local variable bound outside E, and since all parameters to E must appear in the body of E at least once, each time an instance is abstracted it must reduce the number of occurrences of this constant or local variable by one. Thus eventually no more instances of E will remain, and there will be some \( n \) for which abstracts\(_n\) \( x ET \) will be empty, at which point we can terminate the iteration.

\[
\text{abstracts}_0 x ET = \{T\}
\]

\[
\text{abstracts}_{(n+1)} x ET = \{ C \mid B \in \text{abstracts}_n x ET, C \in \text{abstract} x EB \}
\]

We use the function abstract to carry out a single iteration of this procedure; for each subexpression of the current result T it searches all subexpressions of T to check if they are instances of E; if so, it replaces them with x applied to the appropriate arguments. For example, abstract x (\( \lambda y.y + y \)) (1 + (0 + 0)) = \{1 + x 0\}.

Note that only subexpressions that do not already contain the variable x are chosen; this is because we want to guarantee not to change the number of occurrences of x already in T (to fit in with the goal of abstracts\(_n\)). Doing this means that outermost instances are always abstracted first.
abstract \( x E T = \{ \text{replace loc RT} \mid (S, \text{loc}) \in \text{subexps T}, x \notin \text{freevars}(S), R \in \text{instance x E S} \} \)

A subexpression is described as a term together with a sequence representing its position in the term it was taken from (its location). We omit definitions of

\[
\text{subexps :: Exp} \to \{(\text{Exp, Location})\}
\]

\[
\text{replace :: Location} \to \text{Exp} \to \text{Exp} \to \text{Exp}
\]

These functions respectively take a term and returns all its subexpressions and splice a sub-expression into an expression at a given location.

The instance function checks to see if \( S \) is a instance of \( E \), and if so returns the appropriate expression involving \( x \) with which to replace \( S \).

\[
\text{instance x E S} = \{ x (\phi x_1) \ldots (\phi x_n) \mid (x_1, \ldots, x_n) = \text{args } E, \phi \in \text{simplematch (body } E \text{) S} \}
\]

We omit definitions of \text{args} and \text{body}, which simply express \( E \) in the form \( \lambda x_1 \ldots x_n. B \) where \( B \) is not a \( \lambda \)-abstraction and \( n \) may be equal to 0, and return \((x_1, \ldots, x_n)\) and \( B \) respectively.

The function \text{simplematch} is specified as follows. It is simple to implement; it performs pattern matching by a straightforward recursive comparison of the structure of the pattern and the term. The variables \( x_1, \ldots, x_n \) are treated as pattern rather than local variables in the application of \text{simplematch}.

\[
\phi \in \text{simplematch PT} \equiv \phi P \equiv T \land \forall \psi. \psi P \equiv T, \phi \leq \psi
\]

Note that \text{simplematch PT} will always contain 0 or 1 elements.

4 Implementation

As we remarked earlier, we have flexibility as to the order in which we apply \text{resolve} to rules, and thus it makes sense to delay those that will be slow for as long as possible. The most expensive part of this algorithm by far is the function \text{abstracts}, and so a good implementation will only process rules where the pattern is an application with a flexible head when no others remain.

The implementation of \text{abstracts} itself can be improved from our somewhat abstract description. We have used sets throughout to represent lists of results; a concrete program would have to implement these somehow and make certain to remove duplicates where appropriate.

In fact, our restrictions on the pattern ensure that for rules of the form \( F E \to T \) with \( F \) flexible, we have that the body of \( E \) must contain a constant or externally bound local variable. This means that we can always identify all the possible instances of \( E \) in \( T \) immediately (since there can be at most one per occurrence in \( T \) of the constant or local variable from \( E \)). Therefore, instead of going through the iterative procedure given earlier which searches for and abstracts instances one at a time, we can first find all the possible instances
and then carefully work out which combinations of these will produce a valid result before abstracting each combination to produce a set of results; special care needs to be taken to handle the problem of overlapping instances described earlier. This approach also has the advantage that it will not produce duplicate expressions, and so there is no need to add an explicit check for them.

5 Discussion

The use of higher-order matching means that many quite complex program transformations can be expressed as easy to understand rewrite rules. Apart from our own system MAG [7], systems such as KORSO [11] make use of it, while Ergo [16] uses higher-order unification. However, there do also exist many successful transformation systems that avoid it completely, for example Kids [17]. The two standard objections to its use relate to efficiency (even second-order matching is known to be NP-hard [18]), and the need to impose a specific typing discipline. Our algorithm operates on untyped terms, which eliminates the second issue; the first is significant but thus far we have been able to obtain adequate performance for our needs with an implementation that still has plenty of room for improvement.

When we first produced MAG we gave a number of examples of its use. At this time we only had the one-step matching algorithm available and we were thus forced to manually specialise the fusion rules for the more complicated examples so that the derivations could be carried out. With this algorithm we have removed the need to do this in all of the accumulating parameter optimisations described there, and indeed have successfully applied it to yet more complicated problems such as the longest path-sequence problem [2].

Other examples of optimisations MAG is able to apply include cat-elimination in various contexts such as “fast reverse” and the post-order traversal of a tree. Various tree traversals are also susceptible to this form of transformation; in addition to the “minimum depth” example described earlier, fast versions of programs to calculate the breadth-first traversal and to label nodes of a tree with their depth can be derived. A rather more complicated tree processing algorithm, $\alpha\beta$-pruning [4], can also be calculated from the inefficient specification.

We have not yet found an example of such a derivation (i.e. one that involves adding accumulating parameters to a recursive traversal of an inductive datatype) that MAG cannot handle using a standard fusion rule and a combination of the one-step and two-step matching algorithms. We would however expect that significantly more complex problems would be more difficult to handle because of the rather primitive exhaustive rewriting process used in the side calculations. We also anticipate some difficulties with complex problems involving mutually recursive datatypes, because fusion rules on these types have some side conditions which give rise to patterns that violate our restriction, but may in some cases require results not found by the one-step algorithm.

Of course, adding accumulating parameters is just one kind of optimising transformation. Another major class of transformations that we do not consider
here is tupling, i.e. the maintenance of additional useful values in the result of the function in question rather than in the arguments. A simple example of this is the program to calculate the Fibonacci numbers; written the completely naive way this will take exponential time to run, but if we keep track of both the current result and the previous result, this becomes linear time. However, in order to carry out the required derivations automatically our matching algorithm needs to be able to synthesise functions that take tuples as arguments, which the two-step algorithm cannot do. This is an issue we hope to address in future work.

The naming of our algorithms naturally gives rise to the question “what about \( n \)-step matching?” While it should in theory be possible to generalise the step and twostep functions appropriately, there are two reasons why this is unlikely to be worthwhile. Firstly, and most importantly, we know of no practical use for an algorithm that returns more results than two-step matching. Secondly, there was a significant jump in complexity between the one-step and two-step algorithms; extending our algorithm further and in particular finding a suitable set of restrictions on the pattern to make the set of results finite would most likely be very difficult.

It should be noted that our approach to automatic program transformation does not decrease the amount of insight required of a programmer; it is still incumbent on him or her to spot the opportunity for an optimisation and to specify the appropriate rewrite rules to allow this optimisation to be applied. Also, if invalid rewrite rules are chosen, the result will be an incorrect program. Our system removes much of the drudgery associated with the derivation, and allows the insights to be recorded in their original form. In addition, the source-to-source nature of our transformations means that an iterative approach to finding the appropriate set of rewrite rules is possible; the programmer can make a first guess at the correct set, look at the details of the calculation this produces, and modify the set appropriately before trying again.

6 Acknowledgements

The anonymous referees made several helpful comments on an earlier version of this paper. Ganesh Sittampalam would like to thank Microsoft Research for its generous financial support.

References


Appendix

Minimum Depth Derivation

We refer throughout to the automatic derivation of the efficient program for calculating the minimum depth of a binary tree. Here we show how this derivation
was done with MAG\[7\], a transformation system within which we have implemented our algorithm.

The input to the system is the following file, which specifies the rewrite rules we want to use. Of particular interest are the first, which specifies the optimisation we want applied, and the last, which is the fusion rule on binary trees. All the other rules contain the facts about the functions involved that will be needed to carry out the necessary side calculations when applying the fusion rule.

To specify the optimisation, we define the function $md$ that takes the accumulating parameters we want in terms of the inefficient $mindepth$; in addition we “tell” MAG that we want to carry out tree fusion on the variable $t$ by replacing it with a tree fold that is equivalent to the identity function; this fold acts as a “seed” to the fusion. This means that it is not necessary to provide the $mindepth$ program already expressed as a fold.

The fusion rule has side conditions which are universally quantified. Since functions are equal if and only if they are equal for all values of their arguments, we can express this by converting the universal quantifiers into $\lambda$-abstractions on each side of the equation. Note that we could make MAG automatically derive the appropriate fusion rule from the datatype definition, which would reduce the user effort required to specify the transformation.

We have omitted a second input file which is required to provide type information to MAG.

$$\text{md : md t d m = min (mindepth (foldbtree Bin Leaf t) + d) m;}
$$

$$\text{plusunit: 0+a = a;}
$$

$$\text{plusassoc: (a+b)+c = a+(b+c);}
$$

$$\text{minassoc: min (min a b) c = min a (min b c);}
$$

$$\text{cutmin: min (min mq mr + s) c =}
$$

\[
\begin{align*}
& \text{if } s\geq c \\
& \text{then } c \\
& \text{else min (min (mq +s) (mr+s)) c;}
\end{align*}
\]

$$\text{mindepth0: mindepth (Leaf a) = 0;}
$$

$$\text{mindepth1: mindepth (Bin x y) = min (mindepth x) (mindepth y) + 1;}
$$

$$\text{treefusion: h (foldbtree plus f t) = foldbtree times g t,}
$$

\[
\begin{align*}
& \text{if } \{ \text{ \( b \rightarrow h (f b) = \backslash b \rightarrow g b; \)} \\
& \text{ \( x y \rightarrow h (plus x y) = \backslash x y \rightarrow times (h x) (h y) \}\}
\end{align*}
\]

The following derivation is produced by asking MAG to exhaustively rewrite the expression $md$. It should be noted that the final result obtained is expressed in terms of $foldbtree$; it would be a relatively simple matter to inline the definition of this function and thus produce a rather more readable program with the recursion made explicit.
md
= { md }
  ( \ a b -> min (mindepth (foldbtree Bin Leaf a) + b))
= { treefusion
    ( \ b c -> min (mindepth (Leaf b) + c))
= { mindepth0 }
    ( \ b c -> min (0 + c))
= { plusunit }
    ( \ b -> min)

  ( \ b c d -> min (mindepth (Bin b c) + d))
= { mindepth1 }
    ( \ b c d -> min ((min (mindepth b) (mindepth c) + 1) + d))
= { plusassoc }
    ( \ b c d -> min (mindepth b) (mindepth c) + (1 + d)))
= { cutmin }
    ( \ b c d ->
      ( \ e ->
        if a >= e then e else min (min (mindepth b + a)
          (mindepth c + a)) e
      )
    where a = 1 + d
    )
= { minassoc }
    ( \ b c d ->
      ( \ e ->
        if a >= e then e else min (mindepth b + a)
          (min (mindepth c + a) e)
      )
    where a = 1 + d
    )
}
foldbtree ( \ d e f ->
  ( \ g -> if a >= g then g else d a (e a g))
    where a = 1 + f
  )
  ( \ h -> min)
Dynamic Partial Evaluation

Gregory T. Sullivan
Artificial Intelligence Laboratory
Massachusetts Institute of Technology
gregs@ai.mit.edu

Abstract. Dynamic partial evaluation performs partial evaluation as a side effect of evaluation, with no previous static analysis required. A completely dynamic version of partial evaluation is not merely of theoretical interest, but has practical applications, especially when applied to dynamic, reflective programming languages. Computational reflection, and in particular the use of meta-object protocols (MOPs), provides a powerful abstraction mechanism, providing programmatic “hooks” into the interpreter semantics of the host programming language. Unfortunately, a runtime MOP defeats many optimizations based on static analysis (for example, the applicable methods at a call site may change over time, even for the same types of arguments). Dynamic partial evaluation allows us to apply partial evaluation techniques even in the context of a meta-object protocol. We have implemented dynamic partial evaluation as part of a Dynamic Virtual Machine intended to host dynamic, reflective object-oriented languages. In this paper, we present an implementation of dynamic partial evaluation for a simple language – a lambda calculus extended with dynamic typing, subtyping, generic functions and multiple dispatch.

1 Introduction

Our goal is the efficient implementation of dynamic, higher-order, reflective object-oriented languages. Language features that we must support include dynamic typing, runtime method definition, first class types (with subtyping), first class (and higher order) functions, and reflection.

The concept of a meta-object protocol (MOP) [KdB91] subsumes many of the above-listed features. A MOP is based upon computational reflection [Mae87] – giving a program access to its internal structure and behavior and allowing programmatic manipulation of that structure and future behavior. A MOP entails that the entities being returned by reflective operations and manipulated programmatically be first class – thus the need for first class functions, types, classes, etc. We want to at once support the power and abstraction of meta-object protocols while at the same time providing efficient execution, especially when MOP-related features are not being used. For example, if the MOP provides programmer control over method dispatch, but the programmer has not used that feature, the implementation should not instrument every method call.
in order to support the unused abstraction. The research described in this paper is part of a project to implement a Dynamic Virtual Machine (DVM) – a VM well-suited to hosting dynamic, reflective languages. The small formal language $\Lambda_{\text{DVM}}$ presented in this paper is a simplified version of the DVM’s native language, DVML.

Partial evaluation [JGS93] is a general technique for specializing parts of a program with respect to known values. For example, if a function has multiple arguments, and that function is called frequently with the same values for some of the arguments, it may be worthwhile to create a specialized version of that function for those common argument values. Within the body of the specialized version of the function, we may be able to optimize away many computations that depend solely on the values of the arguments against which we are specializing.

Experience has shown that many of the elements of a program exposed and made mutable by a MOP (e.g. the class structure, methods of a virtual function, method dispatch algorithm, etc.) in fact change only rarely at runtime. Thus we may guardedly treat these aspects of an application as constant and then apply partial evaluation techniques to remove computations that depend on these mostly-constant program properties.

2 Related Work

A number of researchers have been drawn to using partial evaluation techniques to eliminate overhead due to reflective operations. This is not surprising, as a recurrent theme in partial evaluation research has been the elimination of “interpretive overhead” (especially by using self application), and reflective operations can be viewed as exposing an interpreter semantics to programs.

Masuhara et al. [MMAY95, MY98] use partial evaluation to eliminate interpretive/reflective overhead in an object-oriented concurrent language. Partial evaluation is performed as part of compilation, and impressive results are recorded, with nearly all interpretive overhead removed in some cases. A limitation of their system is that all possible effects on meta-level functionality must be known at compile time. For example, if modification to the evaluation of variables will be effected at runtime, which modifications must be known at compile time.

In [BN00], Braux and Noyé use partial evaluation techniques to eliminate reflection overhead in Java [GJSB00]. Java provides limited reflection functionality, but no fine-grained MOP to manipulate the features covered by reflection. For example, a Java program may query what methods belong to a class but may not add or remove a method. Java does support coarse-grained redefinition via dynamic class loading. The partial evaluation rules presented in [BN00] are specific to Java and would not work with a different language or runtime.

Online partial evaluation (e.g. [Ruf93]) residualizes with respect to actual values, as does dynamic partial evaluation. The machinery and challenges of dynamic partial evaluation parallel those of online partial evaluation, using runtime
analogs of the structures maintained by online partial evaluation (e.g. generic functions for polyvariant specialization).

Runtime partial evaluation, as in [CN96, VCC97], defers some of the partial evaluation process until actual data is available at runtime. However, the scope and actions related to partial evaluation are largely decided at compile time. Dynamic partial evaluation goes further, deferring all partial evaluation activity to runtime.

The technique of specializing a function on finer types than originally declared, in a language with dynamic dispatch, has been pursued by several research efforts – notably the Self [Cha92] and Cecil [DCG95] projects. In [VCC97] and [SLCM99], Volanchi, Schultz, et al. use declarations and partial evaluation to achieve similar specialization for object-oriented programs. The implementation of dynamic partial evaluation presented in this paper also produces specialized versions of functions.

The runtime partial evaluation in [VCC97] includes the notion of guards against future violation of invariants, and dynamic partial evaluation against “likely invariants” also requires such guards. The idea of optimistic optimization with respect to quasi-invariants has been pursued by Pu et al. in the Synthesis kernel [PMI88] and then Synthetix [PAB+95] projects in the context of operating systems.

3 Overview

We present an overview of the main concepts used in this paper, including dynamic partial evaluation, generic functions, and multiple dispatch.

3.1 Dynamic Partial Evaluation

Dynamic partial evaluation happens as a side-effect of evaluation. At runtime, an expression is evaluated with respect to an environment that contains both the usual dynamic bindings of identifiers to values, and also static bindings from identifiers to types. The static component of the environment corresponds to the symbolic environments of compile-time partial evaluation. In addition to producing a value for the expression, dynamic partial evaluation produces a residual version of the original expression based on the types in the environment. The folding that occurs to produce a residual expression is the same as in online partial evaluation – if the environment indicates that an identifier maps to a fully static value (i.e. has a singleton type), then an expression based on that identifier may be folded. Optimization may occur if a value is not fully static, but its type is known. For example, if accurate types are known for argument values before a call, dynamic type checks may be avoided.

Note that dynamic partial evaluation does not suffer from the “infinite unfolding” issues of static partial evaluation. Because dynamic partial evaluation happens during evaluation, partial evaluation only loops if the application loops.
While dynamic partial evaluation is defined at the expression level, control and collection of the results of dynamic partial evaluation happen at the function level. For example, suppose a function \( f(\text{int } x, \text{ Object } y) \) is called repeatedly with a value of 42 for \( x \) and with (different) instances of the \texttt{Point} class for \( y \). For one such invocation of \( f \), the decision is made to evaluate the body with dynamic partial evaluation enabled. For the duration of this call to \( f \), every expression is evaluated in an environment that maps \( x \) and \( y \) to both their actual concrete values (42 for \( x \), some \texttt{Point} instance for \( y \)), and also to the types \( \texttt{eq}(42) \) and \texttt{Point}. When the execution of \( f \)'s body is complete, we have both a concrete value for this call, and also a new version of \( f \)'s body expression, specialized to the signature \( (\texttt{eq}(42), \texttt{Point}) \). Within the body of the specialized version, we will have performed optimizations assuming that \( x \) is 42 and that \( y \) is an instance of the \texttt{Point} class. The new, specialized version of \( f \) is then added to \( f \)'s \textit{generic function} and will be selected whenever \( f \) is called with its first argument 42 and its second argument an instance of \texttt{Point}.

Dynamic partial evaluation is intended to be used in conjunction with more static techniques. However, the dynamic features of the languages we are targeting often preclude optimizations based on static analysis, and dynamic partial evaluation gives us a valuable tool for optimizing in the face of extreme dynamism.

### 3.2 Generic Functions and Multiple Dispatch

The virtual machine in which we have implemented dynamic partial evaluation provides \textit{generic functions} and \textit{multiple dispatch}. Generic functions and multiple dispatch are used not just to model corresponding features in source programming languages, but are also an integral part of our implementation of dynamic partial evaluation. The key insight is that adding specialized methods to a generic function at runtime corresponds to \textit{polyvariant specialization} in compile-time partial evaluation, and the compile-time notion of \textit{sharing} is handled at runtime via multiple dispatch.

A \textit{generic function} is a set of “regular” functions and selection criteria for choosing one of those functions (or signalling an error) for any given tuple of arguments. In a \textit{single dispatch} language, such as C++ or Java, a generic function corresponds to a virtual method, and the selection criteria is to find the method defined in the class nearest above the receiver’s concrete class in the class hierarchy. \textit{Multiple dispatch} generalizes single dispatch in that more than one argument may be used in the method selection process. For example, we may define a generic function \texttt{foo} consisting of the following methods:

```c
void foo(A this, A that);  // method 1
void foo(B this, B that);  // method 2
void foo(C this, B that);  // method 3
```

Suppose that \( B \) is a subclass of \( A \) and \( C \) is also a subclass of \( A \) (and all classes are instantiable). Then the following sequence of code:

---

1 The notation \( \texttt{eq}(v) \) denotes the \textit{singleton type} containing exactly one value, namely \( v \).
A anA = new A(); B aB = new B(); C aC = new C();
foo(aC, anA); foo(aB, aC); foo(aB, aB); foo(aC, aB);
invokes methods 1, 1, 2, and 3 in order.

Support for generic functions and multiple dispatch serves two distinct purposes in our system. First of all, we are interested in supporting languages with “interesting” dispatch mechanisms, including multiple dispatch such as in Dylan and CLOS. Secondly, it is via the generic function and multiple dispatch mechanisms that we both cache and also invoke specialized versions of functions at runtime. Consider the first call to foo, above. If the first call, foo(aC, anA), is executed with dynamic partial evaluation enabled, we may get a new version of method 1,

// (specialized version of method 1)
void foo(C this, A that); // method 4

Within the body of the new method, dynamic dispatch based on, or dynamic type checking of, the this argument may now be optimized under the assumption that this is an instance of class C. After adding the new method to the foo generic function, later calls to foo with arguments of class C and class A will resolve to this newly created method.

4 \(A_{DVM}\), A Dynamically-Typed Lambda Calculus with Subtyping and Generic Functions

To clarify the mechanism of dynamic partial evaluation, we introduce \(A_{DVM}\), a dynamically-typed lambda calculus with subtypes and generic functions. Figure 1 gives an operational semantics for this simple functional language. \(A_{DVM}\) is modeled after, but much simpler than, DVML, the “native” language of the Dynamic Virtual Machine. Among other things, DVML supports recursive functions, predicate types [EKC98], and more complicated function signatures. The syntax of \(A_{DVM}\) is given by the following grammar:

\[
\text{Exp ::= } x \mid n \mid (\text{if Exp Exp Exp}) \\
\quad \mid (\text{call Exp Exp +}) \mid (\text{gf-call Exp Exp +}) \\
\quad \mid (\text{lambda ([x Exp]+) Exp . Exp})
\]

where \(x\) ranges over identifiers and \(n\) ranges over integers. Note the Exp phrases in lambda expressions for specifying the argument types and result type of a function.

The evaluation relation \(\Rightarrow\) takes an expression \(e\), an environment \(\rho\), and a type \(\tau\) to an extended value \(V\). Extended values are triples of a tagged value \(v\), an expression, and a type. Evaluation must satisfy the following constraints (read \(\simeq\) as “destructures into”):

\[
[\Rightarrow \text{ constraints }] \quad \text{If } [e] \rho \tau \Rightarrow V \simeq \langle v, e', \tau' \rangle_{\text{val}}, \text{ then}
\]

1. \(v : \tau'\), and
2. \(\tau' \leq \tau\).
\( \Rightarrow \subseteq (\mathsf{Exp} \times \mathsf{Env} \times \mathsf{Type}) \times \mathsf{ExtValue} \)

\( \mathsf{ExtValue} = \mathsf{TagValue} \times \mathsf{Exp} \times \mathsf{Type} \)

\( e \) refers to the expression being evaluated in the current rule, “-” means “don’t care”

\( \mathcal{F} \) is a \textit{finish} function defined elsewhere

\( v = \text{some-operation} \) binds \( v \) to the result of \textit{some-operation}.

\( v \simeq e' \) deconstructs value \( v \) into its component parts.

\[
\begin{align*}
\rho([x]) &= V \simeq (v, -, -)_{\mathsf{val}} ; \quad \text{check}(v, \tau) \\
[x] &\rho \tau \Rightarrow \mathcal{F}(e, \langle V, \rho, \tau \rangle) \\
\end{align*}
\]

\[
\begin{align*}
[e_0] &\rho \tau_{\text{bool}} \Rightarrow V_0 \simeq (\text{true}, -, -)_{\mathsf{val}} \\
[e_1] &\rho \tau \Rightarrow V \\
([\text{if } e_0 \ e_1 \ e_2]) &\rho \tau \Rightarrow \mathcal{F}(e, \langle V, V_0, \rho, \tau \rangle) \\
[e_0] &\rho \tau_{\text{fun}} \Rightarrow V_f \simeq (\text{closure}([x_1], \ldots, [x_n]), \langle \tau_{\text{arg}_1}, \ldots, \tau_{\text{arg}_n}, \tau_{\text{res}}, \ [e_f], \rho_f \rangle, -, -)_{\mathsf{val}} \\
[e_1] &\rho \tau_{\text{arg}} \Rightarrow V_i \simeq (v_i, [e'_i], \tau_i)_{\mathsf{val}} ; \ i \in 1 \ldots n \\
[e_f] &\rho_f[x_i \mapsto \langle v_i, [e'_i], \text{glb}(\tau_i, \tau_{\text{arg}})\rangle_{\mathsf{val}}] ; \ \text{glb}(\tau, \tau_{\text{res}}) \Rightarrow V, \ i \in 1 \ldots n \\
\end{align*}
\]

\[
\begin{align*}
([\text{call } e_0 \ e_1 \ldots e_n]) &\rho \tau \Rightarrow \mathcal{F}(e, \langle V, V_f, \langle V_1, \ldots, V_n \rangle \rangle, \rho, \tau) \\
[e_0] &\rho \tau_{\text{mf}} \Rightarrow V_g \simeq (\text{generic}(ms, \langle \tau_{g_1}, \ldots, \tau_{g_n}, \tau_{\text{res}} \rangle), -, -)_{\mathsf{val}} \\
[e_1] &\rho \tau_{g_i} \Rightarrow V_i \simeq (v_i, [e'_i], \tau_i)_{\mathsf{val}} ; \ i \in 1 \ldots n \\
\text{find-mam}(ms, \langle V_1, \ldots, V_n \rangle) &\simeq (V_f, \text{static}?) \\
V_f &\simeq (\text{closure}([x_1], \ldots, [x_n]), \langle \tau_{\text{arg}_1}, \ldots, \tau_{\text{arg}_n}, \tau_{\text{res}}, [e_f], \rho_f \rangle, -, -)_{\mathsf{val}} \\
\text{specialize?}, \langle \text{spec-type}_1, \ldots \rangle &\Rightarrow \text{choose-specialization}(V_g, V_f, \langle V_1, \ldots, V_n \rangle) \\
\text{arg-type}_{e_i} &\simeq (\text{specialize?} \Rightarrow \text{spec-type}_i ; \ \text{glb}(\tau_i, \tau_{\text{arg}})) ; \ i \in 1 \ldots n \\
[e_f] &\rho_f[x_i \mapsto \langle v_i, [e'_i], \text{arg-type}_{e_i} \rangle_{\mathsf{val}}] \ \text{glb}(\tau, \tau_{\text{res}}) \Rightarrow V, \ i \in 1 \ldots n \\
\end{align*}
\]

\[
\begin{align*}
([\text{gf-call } e_0 \ e_1 \ldots e_n]) &\rho \tau \Rightarrow \mathcal{F}(e, F\mathsf{-vals}, \rho, \tau), \text{ where} \\
F\mathsf{-vals} &\simeq (V, V_g, \langle V_1, \ldots, V_n \rangle, V_f, \text{static?}, \text{specialize?}, \langle \text{spec-type}_1, \ldots \rangle) \\
\end{align*}
\]

\[
\begin{align*}
[e_{\tau_i}] &\rho \tau_{\text{type}} \Rightarrow V_{\tau_i} \simeq (v_{\tau_i}, -, -)_{\mathsf{val}} ; \ i \in 1 \ldots n \\
[e_{\tau_{\text{res}}}] &\rho \tau_{\text{type}} \Rightarrow V_{\tau_{\text{res}}} \simeq (v_{\tau_{\text{res}}}, -, -)_{\mathsf{val}} \\
v_f &\simeq \text{closure}([x_1], \ldots, [x_n]), \langle v_{\tau_1}, \ldots, v_{\tau_n}, v_{\tau_{\text{res}}}, [e_0], \rho \rangle \\
V_f &\simeq (v_f, e, \text{eq}(v_f))_{\mathsf{val}} ; \quad \text{check}(v_f, \tau) \\
([\lambda x_1 e_{\tau_1}, \ldots, x_n e_{\tau_{\text{res}}} e_0]) &\rho \tau \Rightarrow \mathcal{F}(e, \langle V_f, \langle V_1, \ldots, V_{\tau_n} \rangle, V_{\tau_{\text{res}}} \rangle, \rho, \tau) \\
\end{align*}
\]

Fig. 1. \( \Lambda_{\text{DVM}} \), A Dynamically typed \( \Lambda \) Calculus with Subtyping and Generic Functions

That is, the type \( \tau \) is a \textit{type constraint} that the returned value, \( v \), must satisfy.

A \textit{tagged value} is a value for which the function \textit{type-of} returns a type. In Figure 1, upper case \( V \)’s range over extended values, and lower case \( v \)’s over tagged values. There is syntax for creating integer, boolean, and closure tagged values, and there are predefined functions for creating other tagged values, including types, generic functions, mutable cells, and lists. A type may be one of the predefined types or built from a type constructor – see Figure 2 for some
predefined values. Creating an extended value with tagged value \( v \), expression \( e \) and type \( \tau \) is denoted \( \langle v, e, \tau \rangle_{\text{val}} \).

Types are ordered as follows: all types are subtypes of \( \top \), \( \bot \) is a subtype of all types, subtyping between types constructed using logical connectives is based on implication, function types have the usual contravariant subtyping, a singleton type \( eq(v) \) is a subtype of \( type-of(v) \), and there is a predefined function, \( \text{subtype} \) for creating subtypes of (multiple) other types.

**Predefined Type Values:** \( \top, \bot, \tau_{\text{int}}, \tau_{\text{bool}}, \tau_{\text{fun}}, \tau_{\text{gf}} \) for top, bottom, integers, booleans, functions (closures), and generic functions, respectively.

**Predefined Boolean Values:** \( \text{true}, \text{false} \).

**Predefined Function Values:**
- \( \text{and} \) (conjunct-types), \( \text{or} \) (disjunct-types), \( \text{not} \) (type), \( \text{fun} \) (arg-types, result-type) – build types from other types.
- \( \text{type-of} \) returns the (concrete) type for a given tagged value \( v \).
- \( \text{check} \) \((v, \tau)\) returns true if \( \text{type-of}(v) \) is equal to or a subtype of \( \tau \); otherwise, halts with an error.
- \( \text{static} ? \) \((\text{ext-val})\) For an extended value \( \langle v, e, \tau \rangle_{\text{val}} \), returns true if \( \tau \leq eq(v) \) – that is, if the value is completely static. We use \( \leq \) rather than \( = \) for type comparison because our type system includes conjunctive types \( \tau \& \tau' \) such that \( \tau \leq (\tau \& \tau') \) and \( \tau' \leq (\tau \& \tau') \).
- \( \text{list} \) \((v_1, \ldots, v_n)\), \( \text{closure} \) \((\text{var}, \text{arg-type}, \text{result-type}, \text{body-exp}, \text{closure-env})\), \( \text{generic} \) \((\text{list-of-methods}, \text{arg-type}, \text{return-type})\), \( \text{subtype} \) \((\text{list-of-supertypes})\), \( eq(\text{val})\): constructors for lists, closures, generic functions, subtypes, and singleton types, respectively.
- \( \text{add-method} \) \((\text{generic}, \text{fun})\) adds a function (method) to a generic function.
- \( \text{glb} \) \((\text{list-of-types})\) constructs the greatest lower bound of its type arguments.
- \( \text{find-mam} \) \((\text{list-of-funs}, \text{arg-vals})\) selects the most applicable method given a set of function (“methods”) and a vector of argument values. If there are no applicable methods, \( \text{find-mam} \) halts with a “no applicable methods” error. If there are multiple (non-comparable) most applicable methods, \( \text{find-mam} \) halts with an “ambiguous methods” error. Otherwise, it returns the most applicable method and also a flag indicating whether method selection was static (more on this in Section 4.2).

**Fig. 2.** Predefined values for \( \Lambda_{\text{DVM}} \)

Environments \( \rho \) map from identifiers to extended values. The **dynamic context** is the projection of the environment as a map from identifiers to tagged values (i.e. the tagged value component of the mapped-to extended value). The **static context** is the projection of the environment as a map from identifiers to types (i.e. the type component of the mapped-to extended value).

When evaluation of an expression is complete, the **finish** function \( \mathcal{F} \) is called with all the values relevant to the just-finished evaluation. \( \mathcal{F} \) has type:

\[(\text{Exp}, \text{Vector(Value)}, \text{Env}, \text{Type}) \rightarrow \text{ExtValue}\]

and must of course satisfy \[\Rightarrow \text{con-}\]
\texttt{constraints} – that is, if \( F(e, \langle V, \ldots \rangle, \rho, \tau) = V' \simeq \langle v, e', \tau' \rangle_{val} \), then \( v : \tau \) and \( \tau' \leq \tau \). For simple interpretation, we instantiate \( F \) as the finish function \( F_{simp} \):

\[
F_{simp}(e, \langle V, \ldots \rangle, \rho, \tau) = V
\]

\( F_{simp} \) simply returns the first value in its value vector. Later, we will define a finish function \( F_{pe} \) that implements dynamic partial evaluation.

### 4.1 Discussion of Evaluation Rules

In Figure 1, the symbol \( e \) always refers to the expression being evaluated. A rule subexpression of the form \( v = \text{some-operation} \) binds \( v \) to the result of \text{some-operation} for use elsewhere in the rule. An expression of the form \( v \simeq e' \) deconstructs the value \( v \), binding the variables mentioned in \( e' \). We use the symbol “\(-\)” to indicate that we will not make use of the corresponding component value.

**var-ref**: Evaluation of a variable reference \( x \) looks up the identifier \( x \) in the environment, checks that it satisfies the current type context \( \tau \), and sends the value to \( F \) (which, in the basic interpreter, simply returns the value).

**integer**: Evaluation of a numeric literal \( n \) checks the value \( n \) against the current type context \( \tau \), constructs an extended value with a fully-static type, and then sends the value to \( F \).

**if-true, if-false**: Evaluation of an if expression first evaluates the test expression \( e_0 \) in a boolean type context, producing \( V_0 \). Either the true branch, \( e_1 \), or the false branch, \( e_2 \), is evaluated, depending on the truth value of the tagged value component of \( V_0 \), and then the resulting value is sent to \( F \), along with \( V_0 \).

**call**: Evaluation of a function call first evaluates the function expression \( e_0 \) in a \( \tau_{fun} \) type context. We then destructure the function value (closure) into its bound variables \( \langle [x_1], \ldots, [x_n] \rangle \), argument types \( \langle \tau_{arg_1}, \ldots, \tau_{arg_n} \rangle \), result type \( \tau_{res} \), body \( e_f \), and the closure’s creation environment \( \rho_f \). Then the call’s argument expressions \( e_i, i \in 1 \ldots n \) are evaluated in type contexts of \( \tau_{arg_i} \). For the values to which the function arguments will be bound, we create new extended values with types that are the greatest lower bound of the static type \( \tau_i \) of the argument value and the function’s argument type \( \tau_{arg_i} \) for each argument position \( i \). Next the body of the function, \( e_f \), is evaluated with the appropriately extended closure creation environment and with a type context that is the greatest lower bound of the function’s return type \( (\tau_{res}) \) and the current type context \( (\tau) \). Finally, all relevant values are sent to \( F \).

**gf-call**: Evaluation of a generic function call first evaluates and destructures its generic function argument. A generic function consists of a triple: a set \( ms \) of functions, aka “methods”, a vector of the argument types \( \tau_{g_i} \), and the result type \( \tau_{g_{res}} \). All functions \( closure(\langle [x_1], \ldots, [x_n] \rangle, \langle \tau_{arg_1}, \ldots, \tau_{arg_n} \rangle, \tau_{res}, e_f, \rho_f) \) in \( ms \) must satisfy the following constraints:

1. \( \tau_{arg_i} \leq \tau_{g_i} \) for each index \( i \), and
2. \( \tau_{res} \leq \tau_{g_{res}} \)
The generic function call arguments are then evaluated with respect to the argument types. Then the helper function \textit{find-mam} is called to select the \textit{most applicable method} from the set \(ms\) given the actual argument values \(V_i\). The function \textit{choose-specialization} decides whether or not to produce a new method for this generic function (using the results of dynamic partial evaluation). If so, \textit{choose-specialization} returns \textit{true} and a vector of argument types. If method specialization is not chosen, the arguments are assigned types as in normal function call— the greatest lower bounds of the declared argument types (of the most applicable method) and the static types of the argument values. For simple interpretation, \textit{choose-specialization} always returns \textit{false}. We discuss other scenarios in Section 5.1. The body \(e_f\) of the most applicable method is then invoked as in a normal function call.

\textbf{abstraction:} Evaluation of a lambda expression first evaluates the expressions for the argument types, \(e_{\tau_i}\), and result type, \(e_{\tau_{res}}\), all of which must satisfy the \(\tau_{type}\) type. Then a closure is created, the value is checked against the current type context \(\tau\), and the closure is sent to \(F\).

\section{4.2 Generic Function Method Selection}

The helper function \textit{find-mam}(\(ms, \langle V_1, \ldots, V_n \rangle\)) first finds the subset of \(ms\) that are applicable given the argument values, \(V_i \simeq (v_i, e'_i, \tau_{i})_{val}:\)

\[ms_{app} = \{ f \mid f \in ms \& f \simeq \text{closure}(\langle [x_1], \ldots, [x_n] \rangle, \langle \tau_{arg_1}, \ldots, \tau_{arg_n} \rangle, \tau_{res}, e_f, \rho_f)\]

\& \text{check}(v_i, \tau_{arg_i}), i \in 1 \ldots n\}

If \(ms_{app}\) is empty, a “no applicable method” error is flagged and execution halts. Next a set of candidates for the most applicable method is derived (ideally a singleton set):

\[mams = \{ f \mid f \in ms_{app} \]
\& \text{closure}(\langle [x_1], \ldots, [x_n] \rangle, \langle \tau_{arg_1}, \ldots, \tau_{arg_n} \rangle, \tau_{res}, e_f, \rho_f)\]
\& (\not\exists f' \in ms_{app} \text{ s.t.}

\[f' \simeq \text{closure}(\langle [x'_1], \ldots, [x'_n] \rangle, \langle \tau'_{arg_1}, \ldots, \tau'_{arg_n} \rangle, \tau'_{res}, e'_f, \rho'_f)\]
\& \tau'_{arg_i} \leq \tau_{arg_i} \text{ for any } i \in 1 \ldots n\})

For any two applicable methods in \(ms_{app}\), if the argument types of one of the methods are all \(\leq\) the corresponding argument types of the other, the second (less specific) method is removed from consideration. It is not allowed for two methods in a generic function to have identical argument type vectors. The set of most applicable method candidates, \(mams\), consists of applicable methods each of which has an argument type vector that is incomparable to the argument type vector of any other method in \(mams\). If the set of most applicable method candidates, \(mams\), has exactly one element, that element is returned; otherwise, an “ambiguous methods” error is signalled and execution halts.

\textit{find-mam} must also keep track of whether the selection of the most applicable method can be accomplished using only the type information in the extended value tuples. For example, suppose the relevant type hierarchy is \(\tau_2 \leq \tau_1\), the single argument value \(V\) has a concrete type of \(\tau_2\), and there are methods for
the generic specialized on $\tau_2$ and $\tau_1$. If the static type of the argument value is $\tau_1$ (that is, $V \simeq \langle v, -, \tau_1 \rangle_{val}$, and $\text{check}(v, \tau_2)$), the choice of method cannot be determined statically (based on the static type alone). However, if the static type is $\text{eq}(v)$ or $\tau_2$, then the method selection is static.

5 Instrumentation for Dynamic Partial Evaluation

Recall that extended values are triples $\langle v : \text{TaggedValue}, e : \text{Exp}, \tau : \text{Type} \rangle_{val}$. In the basic evaluator, only the tagged value component, $v$, is explicitly used. When dynamic partial evaluation is in effect, the expression and type components of an extended value become meaningful. In particular, the following holds (we write $\Rightarrow_{\text{pe}}$ to indicate $\Rightarrow_{\mathcal{F}_{\text{pe}}/\mathcal{F}}$ and $\Rightarrow_{\text{simp}}$ to indicate $\Rightarrow_{\mathcal{F}_{\text{simp}}/\mathcal{F}}$, we use a dash ($-$) for values we do not care about, and we write $v : \tau$ for $\text{check}(v, \tau) = \text{true}$):

**[dpe constraints]** If $e \rho \tau \Rightarrow_{\text{pe}} \langle v, e', \tau' \rangle_{val}$, then
1. $v : \tau' \leq \tau$ (that is, $[\Rightarrow \text{ constraints}]$).
2. $e \rho \tau \Rightarrow_{\text{simp}} \langle v, -, - \rangle_{val}$, and
3. For every environment $\rho'$ that statically matches $\rho$,
   if $e \rho' \tau'' \Rightarrow_{\text{simp}} \langle v', -, - \rangle_{val}$, then
   (a) $e' \rho' \tau'' \Rightarrow_{\text{simp}} \langle v', -, - \rangle_{val}$, and
   (b) $v' : \tau'$

An environment $\rho'$ statically matches an environment $\rho$ if $\text{dom}(\rho) \subseteq \text{dom}(\rho')$ and for all $x \in \text{dom}(\rho)$, if $\rho(x) = \langle -, -, \tau \rangle_{val}$, then $\text{check}(\rho'(x), \tau) = \text{true}$. That is, the types of the bound variables match, though the values may be different.

The statement $[\text{dpe constraints}]$ above specifies that if $e \rho \tau \Rightarrow_{\text{pe}} \langle v, e', \tau' \rangle_{val}$, then the value $v$ is of type $\tau'$ and evaluating the expression with the simple evaluator $\Rightarrow_{\text{simp}}$ will return the same value $v$. Furthermore, evaluating the residualized expression $e'$ in a statically matching environment $\rho'$ will produce the same value as evaluating the original expression $e$ in $\rho'$. In other words, any optimizations that were done to produce $e'$ from $e$ depended only on the types of the values in the environment $\rho$ – that is, the static context. Finally, all values produced by evaluating $e'$ in any statically matching environment will be of type $\tau'$.

5.1 $\mathcal{F}_{\text{pe}}$ – A Finish Function That Implements Dynamic Partial Evaluation

We present a finish function $\mathcal{F}_{\text{pe}}$ that implements dynamic partial evaluation – that is, $\mathcal{F}_{\text{pe}}$ satisfies $[\text{dpe constraints}]$. $\mathcal{F}_{\text{pe}}$ is defined by structural induction on its expression argument. In the following, we present each case for $\mathcal{F}_{\text{pe}}$ along with some discussion.

**Variable Reference:** $\mathcal{F}_{\text{pe}}(\llbracket x \rrbracket, \langle V \rangle, \rho, \tau)$, where $V \simeq \langle v, e', \tau' \rangle_{val}$

- if static?(V) & expressible-as-literal?(v) – if static and a literal

$\langle v, \llbracket v \rrbracket, \tau' \rangle_{val}$ – we can fold to a constant

V
Both the value and the static type for a variable reference come directly from the environment. If the value is completely static and expressible as a literal (that is, an integer or a boolean), the variable reference may be replaced by the corresponding literal expression.

**Literals:** \( F_{\text{pe}}([n], \langle V \rangle, \rho, \tau) = V \)

A literal is always completely static – that is, a given literal expression will always return the same value, no matter in what static context it is evaluated. The reduction relation \( \Rightarrow \) ensures that for literals, \( F \) will be called with a fully-static value.

**Conditionals:**
\[
F_{\text{pe}}([\begin{array}{c} \text{if} \ e_0 \ e_1 \ e_2 \end{array}], \langle V, V_0 \rangle, \rho, \tau), \text{ where } V \simeq \langle v, e', \tau' \rangle_{\text{val}} \text{ and } V_0 \simeq \langle v_0, e'_0, \tau_0 \rangle_{\text{val}}
\]

\[
= \text{ if static?}(V_0) \quad \text{– if test val is a constant,}
\]
\[
V \quad \text{– we can eliminate the conditional}
\]
\[
\text{if } v_0 \quad \text{– otherwise, rebuild the conditional}
\]
\[
\langle v, ([\text{if} \ e'_0 \ e' \ e_2]), \top \rangle_{\text{val}}
\]
\[
\langle v, ([\text{if} \ e'_0 \ e_1 \ e']), \top \rangle_{\text{val}}
\]

The decision whether or not to fold a conditional expression depends on whether or not the test expression is completely static. If the test expression is completely static, the if expression folds away. Otherwise, we rebuild the conditional with the residuals of the test expression and the chosen branch. Note that the static type returned for the value is only \( \top \).

An important optimization for conditionals is the case when \( e_0 \) is a variable reference. In that case, the chosen branch may be evaluated with the environment mapping the test variable to the singleton type of either \( eq(\text{true}) \) or \( eq(\text{false}) \). This is in fact always the case for the Dynamic Virtual Machine, where expressions are all essentially static single assignment form, but we do not present that optimization here.

**Function call:**
\[
F_{\text{pe}}([\begin{array}{c} \text{call} \ e_0 \ e_1 \ldots \ e_n \end{array}], \langle V, V_f, \langle V_1, \ldots, V_n \rangle \rangle, \rho, \tau), \text{ where }
\]
\[
V \simeq \langle v, e', \tau' \rangle_{\text{val}}, V_f \simeq \langle v_f, e'_f, \tau_f \rangle_{\text{val}}, V_i \simeq \langle v_i, e'_i, \tau_i \rangle_{\text{val}}, i \in 1 \ldots n, \text{ and }
\]
\[
v_f \simeq \text{closure}([x_1], \ldots, [x_n], \langle \tau_{\text{arg}1}, \ldots, \tau_{\text{arg}n}, \tau_{\text{res}} \rangle, \rho_f)
\]

\[
= \text{ if static?}(V_f) \quad \text{– if fun is a constant,}
\]
\[
\ldots \text{finish-known-function}\ldots
\]
\[
\quad \text{– here if fun value is not constant}
\]
\[
\langle v, ([\text{call} \ e'_f \ e'_1 \ldots e'_n]), \top \rangle_{\text{val}}
\]

where the code fragment `finish-known-function` is:
\[
(\text{if static?}(V)) \quad \text{– if return val is a constant,}
\]
\[
\langle v, \text{val2exp}(v), \tau' \rangle_{\text{val}} \quad \text{– fold call}
\]
\[
\text{if inline?}(v_f) \quad \text{– if fun is a constant, may choose to inline}
\]
\[
\text{inline-call}(V, V_f, \langle V_1, \ldots, V_n \rangle, \rho, \tau)
\]
\[
\quad \text{– here if fun is constant, but not inlining}
\]
\[
\langle v, ([\text{call} \ e'_f \ e'_1 \ldots e'_n]), \tau_{\text{res}} \rangle_{\text{val}}
\]
Finishing a function call involves choosing one of several options:

1. **Fold the call to either a literal or a variable reference expression.**
   This can only be done if the value of the call is completely static. Folding to a variable reference may involve extending the current environment with a new binding. Residualizing a constant is handled by the function `val2exp`, whose logic is outside the scope of this paper.

2. **Inline the residualized body of the closure at the call site;** preceded by any argument type checks that did not statically succeed. Inlining may also involve extending the current environment, to handle references to variables closed over by the inlined function. Inlining is handled by the function `inline-call`, again outside the scope of this paper.

3. **Replace the call by an unchecked call;** preceded by any argument type checks that did not statically succeed.

4. **Leave the call as is.**

For option 3, the language needs to be extended with operations that do less type checking than base $\Lambda_{\text{DVM}}$, but we do not discuss those in this paper. Note that if the function value is static, but we choose not to inline, we may still use the declared return type of the function for the static type of the returned value.

**Generic Function Call:**

$$
F_{pe}(\text{[\{gf-call } e_0 \ e_1 \ldots \ e_n \}] \ F-\text{vals}, \rho, \tau),
$$

where

$$
F-\text{vals} = \langle V, V_g, \langle V_1, \ldots , V_n \rangle, V_f, \text{static?}, \text{specialize?}, \langle \text{spec-type}_1, \ldots \rangle \rangle,
$$

$$
V \simeq \langle v, e', \tau' \rangle_{\text{vals}},
V_g \simeq \langle v_{g_1}, -,- \rangle_{\text{vals}},
V_g \simeq \text{generic}(ms, \langle \tau_{g_1}, \ldots , \tau_{g_n} \rangle, \tau_{g_{\text{res}}}),
$$

$$
V_f \simeq \langle \text{closure}(\langle [x_1], \ldots , [x_n] \rangle, \langle \tau_{arg_1}, \ldots , \tau_{arg_n} \rangle, \tau_{res}, e_f, \rho_f), -,- \rangle_{\text{vals}},
$$

and

$$
V_i \simeq \langle v_i, e'_i, \tau_i \rangle_{\text{vals}}, i \in 1\ldots n.
$$

If method specialization was chosen, finishing a generic function call adds a new method to the generic function, using the residualized body of the applied method for the closure body, the argument types determined by `choose-specialization`, and other attributes from the original closure.

**Technical note:** Actually, the closure environment is extended with bindings for any static values exposed during inlining that are not expressible as literals and instead bound to fresh variables.

In the Dynamic Virtual Machine, the logic abstracted by `choose-specialization` simply uses programmer-defined rules, in the spirit of [VCC97], to decide when to create specialized versions of generic function methods.  

---

2 In the DVM, method addition is contingent on there being some useful optimization during specialization.
If method specialization was not chosen, then we test whether the generic function value is constant and the most applicable method can be chosen based strictly on the static types of its arguments. In that case, finishing proceeds as in the case for a regular function call when the function argument is static. Recall from Section 4.2 that method selection is considered not static if at least one of the methods of the generic function is not applicable according to the concrete argument types, but is potentially applicable according to the static types of the arguments.

**Abstraction:** \( F_{pe}(\lambda(x_1 \tau_1, \ldots, x_n \tau_n) e_{\tau_{res}} . e_0) \),
\[
\langle V_f, \langle V_{\tau_1}, \ldots, V_{\tau_n}, V_{\tau_{res}} \rangle, \rho, \tau \rangle, \langle V_{\tau_i}, e'_{\tau_i}, - \rangle_{val} \text{ for } i \in 1 \ldots n, \langle V_{\tau_{res}}, e'_{\tau_{res}}, - \rangle_{val} = \langle v_f, \lambda(x_1 e'_{\tau_1}, \ldots, x_n e'_{\tau_n}) e_{\tau_{res}} . e_0 \rangle, \tau_f \rangle_{val}
\]

Lambda abstraction produces a closure value. To finish an abstraction, we return the closure, a rebuilt abstraction expression using the residual expressions from the type expressions, and the singleton type constructed by \( \Rightarrow \).

## 6 Examples of Dynamic Partial Evaluation

We give a few examples of how dynamic partial evaluation works in practice.

### 6.1 A Contrived Example

Suppose we are dynamically partially evaluating the following expression (where \((\text{let } (x \tau) = e_0 \text{ in } e_1)\) is a macro for \(((\lambda(x \tau) \top . e_1) e_0)\):

\[
(\text{let } (a \top) = (\text{if } (> b 0) 3 4) \text{ in } (\text{let } (c \tau_{int}) = (\text{if } (> d 0) 5 6) \text{ in } (\text{gf-call } g a b)))
\]

in the following environment:

\[
b \mapsto \langle 1, 1, eq(1) \rangle_{val}
\]
\[
d \mapsto \langle 1, 1, \tau_{int} \rangle_{val}
\]
\[
g \mapsto \langle \text{generic}(\langle g_1, g_2 \rangle, \langle \top, \tau \rangle, - , eq(g)) \rangle_{val},
\]

where

\[
g_1 = \text{closure}(\langle x, y \rangle, \langle \top, \tau \rangle, \top, \lambda(+ x y), \rho_{g_1})
\]
\[
g_2 = \text{closure}(\langle x, y \rangle, \langle \top, \tau_{int} \rangle, \top, \langle x \rangle, \rho_{g_2})
\]

The identifier \(b\) is statically bound to the integer 1. \(d\) is also bound to 1, but has static type \(\tau_{int}\). The identifier \(g\) is statically bound to a generic function of two methods — one with the most general specializers, and one specialized on integer values for its second argument. The first expression to evaluate is \((> b 0)\). Because \(b\) is completely static, the result value, \(true\) is completely static and the if expression can be folded. The result of the if expression is the fully static value 3. The identifier \(a\) is bound to the value 3, with residual expression 3, and gets static type \(eq(3)\), which is the greatest lower bound (glb) of the declared type \(\top\) and the result type \(eq(3)\) of the expression. The comparison \((> d 0)\) also evaluates to

\[
g \mapsto \langle -, -, eq(g) \rangle_{val} \text{ means that } g \text{ is completely static.}
\]
true, but the value is not completely static because the static type of \( d \) is \( \tau_{int} \). The identifier \( c \) is bound to the value 5, residual expression (if \( (> d 0) 5 6 \)), and gets static type \( \tau_{int} \), which is the glb of the declared type \( \tau_{int} \) and the type of the result of the expression, namely \( \top \). The most applicable method for the call to \( g \) is \( g_2 \). Furthermore, static type \( \tau_{int} \) of variable \( c \) is sufficient to statically select the most applicable method at the call, so the gf-call can be replaced by a simple call to \( g_2 \). When the body of \( g_2 \) is executed, it returns the value of its first argument, \( a \), which is fully static. Because the method can be statically selected, and because the result of the call is a fully static value, the whole gf-call expression can be folded to the literal expression 3. Thus the expression residualizes to:

\[
\text{let } (c \, \tau_{int}) = (\text{if } (> d 0) 5 6) \text{ in } 3
\]

Dead variable elimination may eliminate the now useless let construct.

### 6.2 Example: Dynamic Partial Evaluation of Reflection in Java

In [BN00], Braux and Noyé use partial evaluation techniques to eliminate reflection overhead in Java. The rules they introduce are specific to the reflection API of Java. Dynamic partial evaluation provides a general mechanism that automatically eliminates the reflection overhead addressed by Braux and Noyé. Following is the main example from [BN00]:

```java
public static void dumpFields(Object anObj)
throws java.lang.IllegalArgumentException {
    Field[] fields = anObj.getClass().getFields();
    for (int i = 0; i < fields.length; i++)
        System.out.println(fields[i].getName() + ": " + fields[i].get(anObj));
}
```

If `dumpFields` is called often on a specific class, say `Point`, it is worthwhile to create a specialized version of `dumpFields` specific to `Point`. Assume for now that the `Point` class has no subclasses. Within the specialized version of `dumpFields`, most of the reflection overhead can be folded away – the call to `getFields` is always return the `Point` class, and `getFields` will always return an array containing the `x` and `y` fields. Of course, the values of `x` and `y` are dynamic – that is, they will vary between invocations of the method. After partial evaluation, the specialized method should be something like:

```java
public static void dumpFieldsPoint(Point anObj) {
    System.out.println("x: "+anObj.x);
    System.out.println("y: "+anObj.y);
}
```

We have an implementation of the relevant parts of the Java runtime, and a translation from Java into the Dynamic Virtual Machine (DVM). The above example, in the case that `Point` has no subclasses, folds to DVM code analogous to that given above, and a method specialized on `Point` is added to the `dumpFields` generic function. Thereafter, calls to `dumpFields` with `Point` arguments automatically select the optimized version.
Furthermore, if later calls to \texttt{dumpFields} take place within the context of specializing some other generic function, and the argument is statically bound to \texttt{Point}, the optimized code may be inlined into the calling method, and so on.

7 Optimistic Dynamic Partial Evaluation

As was mentioned in the introduction, we want to optimize with respect to “quasi-invariants” – in particular, elements of the meta-object protocol (MOP) that are technically mutable but rarely modified in practice.

In the Dynamic Virtual Machine, there are two mutable datatypes: cells and generic functions. A cell contains a single value that may be changed, and a generic function may be modified by updating its method list. As dynamic partial evaluation proceeds, each optimization (folding, inlining) notes any cells or generic functions that have been referenced. When a new method is added to a generic function as a result of dynamic partial evaluation, all referenced cells and referenced generic functions are instrumented to undo the optimization if mutated.

In the example from Section 6.1, suppose the expression exists in a method of the generic function named \texttt{h}. After the body of the method finishes, a new version of the method, including the residual expression from the example, is added to \texttt{h}’s method list. In this case, a dependency is recorded between the generic function \texttt{g} and the newly-added method. If at some later point \texttt{add-method} is called on the \texttt{g} generic function, the newly added method is removed from \texttt{h}’s method list.

In the example from Section 6.2, dependencies are created between the generic functions \texttt{getClass}, \texttt{getFields}, \texttt{length}, \texttt{getName}, and \texttt{get} and the specialized version of \texttt{dumpFields}. Adding a new subclass of \texttt{Point} will add new methods to the generic functions \texttt{getFields}, \texttt{getName}, and \texttt{get}, thus removing the specialized method.

In fact, with our current dependency tracking, adding \textit{any} new class to the system will cause the specialized method of \texttt{dumpFields} to be removed, because our level of granularity is only at the generic function level, as opposed to specific tuples of type hierarchies.

In the Dynamic Virtual Machine, cells and generic functions exposed by the MOP are considered “quasi-invariant” and dynamic partial evaluation tracks references to them.

8 Controlling Dynamic Partial Evaluation

Dynamic partial evaluation is subject to the “infinite specialization” problem of polyvariant specialization systems. The Dynamic Virtual Machine associates a set of “specialization rules” with each generic function; each rule includes a predicate and a specialization signature. When a generic function is invoked, and after the most applicable method has been chosen, the specialization rules
of the generic are matched against the values of the method, the argument values, and the argument values’ static types. If a rule matches, it specifies the signature against which specialization should occur. This is similar in spirit to the specialization classes of [VCC97].

Currently, the specialization rules are given by the programmer. To do the example from Section 6.2 we added a rule to the `dumpFields` generic that matches methods specialized to `Object` and produces specialization against the concrete type of the argument.

Our goal is to dynamically generate specialization rules based on profile information, as done in [DCG95].

9 Tracking Side Effects

As for any partial evaluator for an imperative language, dynamic partial evaluation must avoid folding function calls to values when there may be side effects involved. The Dynamic Virtual Machine handles this by threading a `side-effecting` flag through evaluation of an expression. Thus, a function call may return a fully static value, but the call expression cannot be residualized to a constant if there were side-effecting operations involved. For example, a fully static call on fully static argument values that returns a newly-allocated list of those argument values cannot be folded to the list itself, because next time that call should return another newly allocated list (containing the same values). For a function call with a non-static function, and for a conditional with a dynamic test, dynamic partial evaluation must be pessimistic about whether side effects occur. In the Dynamic Virtual Machine, the primitive operations define whether or not they are side-effecting.

10 Conclusions and Future Work

Dynamic partial evaluation is a technique for instrumenting interpretation in order to perform partial evaluation actions as a side effect of evaluation. This is accomplished by interpreting expressions in an environment that maps identifiers not only to values but also to types. The type of a variable can be understood as “how much information dynamic partial evaluation is allowed to assume about this binding.”

Dynamic partial evaluation has been implemented as part of a Dynamic Virtual Machine designed to host dynamic, reflective, higher-order languages with subtyping. In the current implementation, dynamic partial evaluation is always “on” – that is, evaluation always creates residual expressions and tracks static types. We would like to be able to dynamically switch between dynamic partial evaluation and simple interpretation – suffering the overhead of dynamic partial evaluation only when we know we will use the results.

As far as when to enable dynamic partial evaluation of a method, we currently specify rules by hand, in the spirit of [VCC97]. We plan to use dynamically
generated profile data to decide when and where to do dynamic partial evaluation. Note that we are focusing on highly reflective runtime environments, so profile data should be readily available. We also plan on using more sophisticated techniques for deciding when to inline function bodies.

We have not yet addressed the efficiency of multiple dispatch itself, but we intend to follow the lead of Chambers and Chen, [CC99]. The idea is that a generic function call expression is replaced by an inline binary decision tree, with leaves being direct calls to methods. Again, a method with generic calls replaced by decision trees is guardedly added to the generic function. When the generic functions involved are modified at runtime, the specialized version is removed and the original method, with generic function calls, is restored.

At the time of this writing, the two most vexing issues are:

- Deoptimizing methods that have an active call (related to on-stack replacement in Self), and
- Avoiding a bad interaction between newly specialized methods and previously specialized methods. When adding a newly specialized method causes previously specialized methods to be invalidated, we lose.

We are exploring several approaches to both problems.

The ideas of dynamic partial evaluation apply to any level of interpretation, and the $\Lambda_{DVM}$ language and the Dynamic Virtual Machine are fairly high level. The intent is that interpretation, including dynamic partial evaluation, will spend only enough time at this very high level to do optimizations specific to that level – in particular, optimizations with respect to user-defined types. After a flurry of specialization, the goal is to translate to either a low-level virtual machine or to native instructions where further dynamic optimization may take place. The lower level representations of methods (that is, in terms of native instructions rather than DVM instructions) are cached using the same generic function mechanism. When assumptions made during code generation are violated, the native version is replaced by its original high level version.

Acknowledgements

Many thanks to Julia Lawall for carefully reading initial drafts of this paper – the paper is much better thanks to her comments. Thanks also to Mitch Wand for valuable discussions and to the anonymous referees for many insightful comments.

References


Tag Elimination and Jones-Optimality
(Preliminary Report)

Walid Taha¹*, Henning Makholm²**, and John Hughes³

¹ Department of Computer Science, Yale University, New Haven, CT, USA
taha@cs.yale.edu

² DIKU, University of Copenhagen, Copenhagen, Denmark
henning@makholm.net

³ Department of Computing Sciences, Chalmers, Göteborg, Sweden
rjmh@cs.chalmers.se

Abstract. Tag elimination is a program transformation for removing unnecessary tagging and untagging operations from automatically generated programs. Tag elimination was recently proposed as having immediate applications in implementations of domain specific languages (where it can give a two-fold speedup), and may provide a solution to the long standing problem of Jones-optimal specialization in the typed setting. This paper explains in more detail the role of tag elimination in the implementation of domain-specific languages, presents a number of significant simplifications and a high-level, higher-order, typed self-applicable interpreter. We show how tag elimination achieves Jones-optimality.

1 Introduction

In recent years, substantial effort has been invested in the development of both theory and tools for the rapid implementation of domain specific languages (DSLs). DSLs are formalisms that provide their users with notation appropriate for a specific family of tasks at hand. A popular and viable strategy for implementing domain specific languages is to simply write an interpreter for the DSL in some meta-language, and then to stage this interpreter either manually by adding explicit staging annotations (multi-stage programming [16,10,14]) or by applying an automatic binding-time analysis (off-line partial evaluation [6]). The result of either of these steps is a staged interpreter. A staged interpreter is essentially a translation from a subject-language (the DSL) to a target-language[1]. If there is already a (native code) compiler for the target-language, the approach yields a simple (native code) compiler for the DSL at hand.

This paper is concerned with a costly problem which can arise when both the subject- and the meta-language are statically typed. In particular, when

* Funded by a Postdoctoral Fellowship from the Swedish Research Council for Engineering Sciences (TFR), grant number 221-96-403, and by subcontract #8911-48186 from Johns Hopkins University under NSF agreement Grant # EIA-9996430.

** Funded by the University of Copenhagen.

¹ Staging also turns the meta-language to be (additionally) the target-language.
the meta-language is typed, there is generally a need to introduce a “universal
datatype” to represent values. At runtime, having such a universal datatype
means that we have to perform tagging and untagging operations. When the
subject-language is untyped, we really do need these checks (e.g. in an ML
interpreter for Scheme). But when the subject-language is also statically typed
(e.g. an ML interpreter for ML), we do not really need the extra tags: they are
just there because we need them to statically type check the interpreter. When
such an interpreter is staged, it inherits this weakness and generates programs
that contain superfluous tagging and untagging operations. To give an idea of the
cost of these extra tags, here is the cost of running two sample programs (the
factorial function applied to 12 and the Fibonacci function applied to 10) with
and without the tags in them:\(^2\):

<table>
<thead>
<tr>
<th>Term (fully inlined)</th>
<th>fact 12</th>
<th>fib 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup (after tag elimination)</td>
<td>2.6x</td>
<td>1.9x</td>
</tr>
</tbody>
</table>

The table shows that removing the superfluous tags from these two programs
speeds up their execution by a factor of 2.6 and 1.9 times, respectively.

How, then, can we ensure that programs produced by the staged interpreter
do not contain superfluous uses of the universal datatype?

One possibility is to look for more expressive type systems that alleviate the
need for a universal datatype (such as dependent type systems). But it is not
clear that self-interpretation can be achieved in such languages\(^1\). A more
pressing practical concern is that such systems lose decidable type inference,
which is a highly-valued feature of many typed functional programming lan-
guages.

Tag elimination\(^1\) is a recently proposed transformation that was de-
signed to remove the superfluous tags in a post-processing phase. Thus our
approach is to stage the interpreter into three distinct stages (rather than the
traditional two). The new extra stage, called tag elimination, is distinctly differ-
ent from the traditional partial evaluation (or specialization) stage. In essence,
tag elimination allows us to type check the subject program after it has been in-
terpreted. If it checks, superfluous tags are simply erased from the interpretation.
If not, a “semantically equivalent” interface is added around the interpretation.

1.1 Jones-Optimality

The problem of the superfluous tags is tightly coupled with the problem of
Jones-optimal self-interpretation in a statically-typed language. The significance
of Jones-optimality lies both in its relevance to the effective application of the
above strategy when a statically-typed meta-language is used, and the fact that
the problem has remained open for over thirteen years, eluding numerous signif-
icant efforts\(^3\)\(^4\)\(^5\).

Intuitively, Jones-optimality tries to address the problem of whether for a
given meta-language there exists a partial evaluator strong enough to remove

\(^2\) Data based on 100,000 runs of each in SML/NJ.
an entire level of “interpretive overhead” [6, Section 6.4]. A key difficulty is in formalizing the notion of interpretive overhead. To this end, Jones chose to formulate this in the special case where the program being specialized is an interpreter. This restriction makes the question more specific, but there is still the question of what removing a layer of interpretive overhead means, even when we are specializing an interpreter. One choice is to say that the cost of specializing the interpreter to a particular program produces a term that is no more expensive than the original program. This, however, introduces the need for a notion of cost, which is non-trivial to formalize. Another approach which we take here is to say that the generated program must be syntactically the same as the original one. While this requires prohibiting additional reductions, we accept that, as it still captures the essence of what we are trying to formalize.

The relevance of Jones-optimality lies in that, if we cannot achieve it, then staging/partial evaluation will no-doubt produce sub-optimal programs for a large variety of languages. Thus, resolving this problem for statically-typed programming languages means that we have established that statically-typed languages can be used to efficiently implement compilers for a large class of domain specific languages (including, for example, all languages that can be easily mapped into any subset of the language that we consider).

1.2 Contribution and Summary of the Rest of the Paper

This paper shows how tag elimination achieves Jones-optimality, and reports (very briefly) on an implementation that supports our theoretical results. In doing so, this paper extends previous theoretical work [15] by presenting 1) a typed, high-level language together with a self-interpreter for it (needed for Jones-optimality), and 2) a substantially simplified version of tag-elimination. Previous implementation work [7] was in a first-order language.

Section 2 presents a simply-typed programming language that will be used as the main vehicle for presenting the self-interpreter and the proposed transformation. The language has first-order data and higher-order values. We define the new annotations and their interpretations. A specification of a tag elimination analysis is presented in Section 3 as a set of inference rules defined by induction on the structure of raw terms. In Section 4 we summarize the basic semantic properties of tag elimination. In this section, we define the wrapper and unwrapper functions that are needed to define a “fall-back” path for the tag-elimination transformation.

Section 5 reviews interpreters. Section 6 addresses the relation between an interpreter and a staged interpreter, emphasizing the utility of the notion of a translation in this setting. In this section, we show how tag elimination analysis

---

3 In this paper, we present and focus on a specification of the analysis, not an algorithm for carrying it out. We expect that the analysis can be implemented by a total function that yields a result that can be validated by our specification, but at this time, we have not yet established this formally. In this paper, we will often refer to this specification as “the analysis”.
is sufficient to allow us to eliminate superfluous tags from typed staged self-interpreters. In Section 7 we demonstrate the relevance of this result to the problem of Jones-optimal specialization.

2 A Typed Language for Self-Interpretation

First we present a programming language with first-order datatypes and with higher-order values. The types \( \mathbb{T} \) in this language are simply:

\[
\begin{align*}
t &::= D | V | t \rightarrow t
\end{align*}
\]

The type \( D \) is for first-order data (like LISP S-expressions), the type \( V \) is for higher-order values (a universal datatype), and the last production is for function types. We can think of \( V \) as being generated by the following ML declaration:

\[
\text{datatype } V = \text{Fo } f \text{V} \rightarrow \text{V} | \text{Eo } f \text{D}
\]

But we do not need case analysis on \( V \): We will only need value constructors (tagging) and simple destructors (untagging) for our purposes here (writing interpreters). We assume an infinite set of names \( X \) ranged over by \( x \), and that this set includes the special variables “nil, true, false”. The set of expressions \( E \) is defined as follows:

\[
\begin{align*}
s &::= x | (s.s) \quad u &::= \text{car} | \text{cdr} | \text{atom} \? \quad o &::= \text{cons} | \text{equal} \?

e &::= x | e | \lambda x.e | \text{fix } e | \text{if } e \ 1 e \ 2 e \ 3 | \text{E } e | \text{E}^{-1} e | \text{F } e | \text{F}^{-1} e
\end{align*}
\]

The type \( D \) will be inhabited by S-expressions represented by dotted-pairs \( s \). One can use a distinct set for names of atoms, but it causes no confusion to simply use variables here. Note that substitution “does not do anything” with ‘s, or rather, it is the identity. We use a standard type system for this language.

\[
\begin{align*}
\Gamma(x) = t & \quad \Gamma \vdash e_1 : t_1 \rightarrow t_2 \quad \Gamma \vdash e_2 : t_1 \quad \Gamma \vdash e : t_2 \quad \Gamma \vdash x : t_1 \vdash e : t \quad \Gamma \vdash x : t \vdash e_1 e_2 : t_2

\Gamma \vdash \text{fix } x.e_1 e_2 : t_2 \quad \Gamma \vdash \lambda x.e : t_1 \rightarrow t_2 \quad \Gamma \vdash e : t

\Gamma \vdash e_1 : D & \quad \Gamma \vdash e_2 : D & \quad \Gamma \vdash e_2 : t \quad \Gamma \vdash e_1 : D

\Gamma \vdash e : D & \quad \Gamma \vdash e : V & \quad \Gamma \vdash e : V \rightarrow V & \quad \Gamma \vdash e_1 e_2 : V & \quad \Gamma \vdash \text{if } e_1 e_2 : t

\Gamma \vdash \text{E } e : V & \quad \Gamma \vdash \text{E}^{-1} e : D & \quad \Gamma \vdash \text{F } e : V & \quad \Gamma \vdash \text{F}^{-1} e : V \rightarrow V
\end{align*}
\]

\(4\) The fixed-point construct used here seems to worry some readers. In particular, they expect such a construct to either take an additional parameter, or have a type restricted to function types, or both. We choose this form primarily because it keeps the various type-preserving translations simple. It may help the reader to note that any term written in such a language as ours can be easily translated into a language that uses a fixed-point operator restricted to function types. Note, however, that in a language with a fixed-point operator such as ours, variables are not values.
The type system enjoys weakening and substitution properties.

**Lemma 1** (Weakening, substitution).

1. \( \Gamma \vdash e : t_1 \land x \notin \text{FV}(e) \cup \text{dom}(\Gamma) \Rightarrow x : t_2; \Gamma \vdash e : t_1 \)
2. \( \Gamma \vdash e_1 : t_1 \land x : t_1; \Gamma \vdash e_2 : t_2 \Rightarrow \Gamma \vdash e_2[x := e_1] : t_2 \).

**Proof.** All by easy inductions. \(\square\)

A standard **big-step operational semantics** \(\rightsquigarrow : \mathbb{E} \rightarrow \mathbb{E} \) for this language is used:

\[
\begin{array}{llll}
    e_1 \rightsquigarrow \lambda x.e_3 & e_2 \rightsquigarrow v_1 & e_3[x := v_1] \rightsquigarrow v_2 & e_1 e_2 \rightsquigarrow v_2 \\
    \lambda x.e \rightsquigarrow \lambda x.e & e[x := \text{fix } x.e] \rightsquigarrow v & \text{fix } x.e \rightsquigarrow v & e_1 \rightsquigarrow (s_1,s_2) \\
    \text{car } e_1 \rightsquigarrow s_1 & \text{cdr } e_1 \rightsquigarrow s_2 & \text{atom? } e \rightsquigarrow \text{true} & \text{atom? } e \rightsquigarrow \text{false} \\
    e_1 \rightsquigarrow (s_1,s_2) & e_2 \rightsquigarrow s_2 & e_1 \rightsquigarrow s & e_2 \rightsquigarrow s \\
    \text{equal? } e_1 e_2 \rightsquigarrow \text{true} & \text{equal? } e_1 e_2 \rightsquigarrow \text{false} & \text{if } e_1 e_2 e_3 \rightsquigarrow v_2 & \text{if } e_1 e_2 e_3 \rightsquigarrow v_3 \\
    e \rightsquigarrow v & e \rightsquigarrow \mathbb{E} v & e \rightsquigarrow F v & e \rightsquigarrow F v \\
    \mathbb{E} e \rightsquigarrow \mathbb{E} v & \mathbb{E}^{-1} e \rightsquigarrow v & F e \rightsquigarrow F v & F^{-1} e \rightsquigarrow v
\end{array}
\]

This semantics induces a set of values, namely, the largest set of terms on which the semantics is idempotent. The set of **values** \(\mathbb{V} \) is defined as follows:

\[
v := \lambda x.e \mid s \mid \mathbb{E} v \mid F v
\]

Note that \(\mathbb{V} \subset \mathbb{E}\). This containment is one of the reasons why our treatment is often considered “syntactic” (as opposed to “denotational”). We find the containment useful because it allows us to avoid having two similar but still slightly different notions of various concepts, such as typing.

We can refine the type of evaluation to \(\rightsquigarrow : \mathbb{E} \rightarrow \mathbb{V}\). The three basic properties of values are the following:

**Lemma 2** (Values).

1. \( e_1 \rightsquigarrow e_2 \land e_1 \rightsquigarrow e_3 \Rightarrow e_2 = e_3, \text{ and} \)
2. \( e_1 \rightsquigarrow e_2 \Rightarrow e_2 \in \mathbb{V}, \text{ and} \)
3. \( v \rightsquigarrow v. \)

---

\(5\) We could have written \(\mathbb{E} \rightsquigarrow s\) and \(F \lambda x.e\) for the last two cases, but that puts unnecessary restrictions on the untyped language. It also fails to give us “the largest” set on which the semantics is idempotent. In the typed setting, the type system will ensure that typed values will necessarily have the more restricted form.
Proof. All proofs are by simple inductions over the height of the derivation.

The semantics also enjoys type-preservation.

Lemma 3 (Type Preservation). \( \Gamma \vdash e : t \land e \leftrightarrow v \implies \Gamma \vdash v : t \)

Note, however, that it is still possible for some terms to “get stuck” \(^{18}\) in our language, such as in trying to take the tail (\texttt{car}) of an empty list (\texttt{nil}). We will write \(\equiv\) for syntactic equality of terms, up to \(\alpha\) conversion. For semantic equality we will use the largest congruence when termination is observed\(^6\). A context \(C\) is a term with exactly one hole [\]. We will write \(C[e]\) for the variable capture filling of the hole in \(C\) with the term \(e\). Two terms \(e_1, e_2 \in \mathbb{E}\) are observationally equivalent, written \(e_1 \approx e_2\), when for every context \(C\) it is the case that:

\[
(\exists v. C[e_1] \leftrightarrow v) \iff (\exists v. C[e_2] \leftrightarrow v)
\]

2.1 Semantics-Preserving Annotations

The key idea behind the proposed approach to dealing with the interpretive overhead is that the user writes an interpreter which includes some additional annotations that have no effect on the semantics of the program but that do have an effect on what happens to the superfluous tags. Thus, the programmer writes the interpreter in a language of annotated terms. An annotated term \(\hat{e} \in \hat{\mathbb{E}}\) is a term where each occurrence of \(\mathbb{E}, \mathbb{E}^{-1}, \mathbb{F}, \mathbb{F}^{-1}\) is annotated with one of two annotations \(B\):

\[
b ::= k | e
\]

Where \(k\) stands for “keep” and \(e\) stands for “eliminate”. Substitution is defined on annotated terms in the standard manner. Any term can be lifted into an annotated term. Lifting, written, \([\_\_]\) simply annotates every constructor and destructor with the tag \(k\). Lifting is substitutive: \([e_1][x := [e_2]] \equiv [e_1[x := e_2]]\). Lifting types and environments is simply the identity embedding.

Annotated contexts are defined similarly to terms. Lifting on contexts \([C]\) is defined similarly to terms. The evaluation function on terms can be lifted to annotated terms where all the constructs are propagated during a computation without inspecting or making any changes to the annotations (Thus, it’s OK to use a \(k\)-untag operation to remove an \(e\)-tag in this semantics.)

The subject interpretation on annotated terms \(\_|\_ : \hat{\mathbb{E}} \to \mathbb{E}\) simply forgets the annotations, and the target interpretation \(\|\| : \hat{\mathbb{E}} \to \mathbb{E}\) eliminates

\(^6\) Note that, because we have datatypes and some datatype operations can get stuck, one should ideally use a notion of equivalence which distinguishes between getting stuck and diverging. In this paper, we avoid this distinction for the sake of simplicity. Because nowhere in our treatment do we exchange a possibly-stuck term with a possibly-non-terminating term (or the other way around), we expect our results to generalize.
constructs annotated with $e$ and just drops the $k$ annotation from the others. For example, $||F_e (\text{car} x)|| \equiv \text{car} x$. Note that $|[e]| \equiv ||[e]|| \equiv e$, and that both notions of erasure are substitutive. Both notions of erasure are also onto. These facts will allow us to keep reasoning with observational equivalence simple.

The subject interpretation allows us to lift observational equivalence to annotated terms, that is, we will define equivalence on annotated terms as follows:

$$\hat{e}_1 \approx \hat{e}_2 \overset{\text{def}}{\iff} |\hat{e}_1| \approx |\hat{e}_2|$$

This means that, from the user’s point of view, tagging and untagging operations annotated with $e$ or $k$ are semantically the same. Thus, we can really think of these annotations as being purely hints to the tag-elimination analysis that can affect only performance. In this paper, if one “hint” is wrong, no tag elimination will be performed at all. The goal of this paper is not, however, to demonstrate that there is a robust analysis that solves optimality, but rather that there is an analysis at all. (See Makholm \[7\] for some ideas to alleviate this practical problem.)

3 A Specification of a Tag Elimination Analysis

In this section, we present a specification of a new tag-elimination analysis. The analysis will be presented as a type system defining a judgment $\Gamma \vdash e/a$. Intuitively, the judgment says “$a$ describes the type of $e$ before and after the extra tags are eliminated”. Annotated types $\hat{T}$ are basically types carrying names of tags (either $E$ or $F$) in certain positions, and are defined as follows:

$$\hat{t} ::= D \mid V \mid \hat{t} \rightarrow \hat{t} \mid E \hat{t} \mid F \hat{t}$$

We will use two strict subsets ($C \subset \hat{T}$ and $A \subset \hat{T}$) that can identify two special families of refinements:

$$c ::= V \mid E D \mid F (c \rightarrow c)$$
$$a ::= D \mid c \mid a \rightarrow a$$

An annotated type $c$ identifies a subset of values (and terms) of type $V$, and an annotated type $a$ corresponds to legitimate type-specializations of a (subject) value of any type. In the case of the first production for $c$, the subset is the whole set. In the second case, the subset values that have $E$ tags as the outermost (or top most) construct. In the next case, values that have a $F$ tag. For terms, the annotated type identifies certain terms that can either diverge or evaluate to values identified by the annotated type.

In the context of the work of Hughes and Danvy, an annotated type $E D$ can be seen as describing a “type-specialization path” from a value of type $V$ to a value of type $D$. The additional tag information in the annotated type tells
us that we achieve this “type specialization” (semantically) by eliminating an \( E \) tag. The \textbf{subject} \(|a|\) and \textbf{target} \(||a|||\) interpretations of annotated types are:

\[
|D| = D, \quad |c| = V, \quad |a_1 \rightarrow a_2| = |a_1| \rightarrow |a_2|,
\]

\[
||D|| = ||E D|| = D, \quad ||V|| = V, \quad ||F a|| = ||a||, \quad ||a_1 \rightarrow a_2|| = ||a_1|| \rightarrow ||a_2||.
\]

The tag elimination analysis is defined as follows:

\[
\begin{align*}
\Gamma(x) &= a & \Gamma \vdash \hat{e}_1 / a_1 \rightarrow a_2 \\
& \quad \vdash x / a & \Gamma \vdash \hat{e}_2 / a_1 & \rightarrow a_2 \\
& \quad \vdash \hat{e}_1 \hat{e}_2 / a_2 & \Gamma \vdash \lambda x. \hat{e}/a_1 \rightarrow a_2 & \Gamma \vdash \text{fix } x. \hat{e}/a \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash \hat{e} / D & \quad \Gamma \vdash \hat{e} / V & \Gamma \vdash \hat{e} \rightarrow V & \Gamma \vdash \hat{e} / V \\
\Gamma \vdash s / D & \quad \Gamma \vdash u \hat{e} / D & \Gamma \vdash a \hat{e}_1 \hat{e}_2 / a_2 & \Gamma \vdash \text{if } \hat{e}_1 \hat{e}_2 \hat{e}_2 / a_2 \\
& \quad \vdash \hat{e} / D & \Gamma \vdash \hat{e} / V & \Gamma \vdash \text{if } \hat{e} / V \\
\Gamma \vdash \text{E}_k \hat{e} / V & \Gamma \vdash \text{E}^{-1}_k \hat{e} / D & \Gamma \vdash \hat{e} / c_1 \rightarrow c_2 & \Gamma \vdash \hat{e} / (c_1 \rightarrow c_2) \\
\Gamma \vdash \text{E}_e \hat{e} / \text{D} & \Gamma \vdash \text{E}^{-1}_e \hat{e} / D & \Gamma \vdash \hat{e} / \text{E}(c_1 \rightarrow c_2) & \Gamma \vdash \hat{e} / \text{E}^{-1}(c_1 \rightarrow c_2)
\end{align*}
\]

The first two lines of the type system are completely standard. The last line introduces new rules that assign special annotated types for tagging and untagging operations annotated with “eliminate” annotations.

This type system also enjoys weakening and substitution properties, and the semantics also enjoys an \textbf{analag of type-preservation} on closed terms. The analysis \textit{includes} the type system, in that any type judgment \( \vdash e : t \) has a \textbf{canonical} corresponding analysis judgment \( \vdash [e]/t \). We can also establish stronger properties of the analysis:

\begin{lemma}[Double Typing].
\[
\vdash \hat{e} : |a| \iff \vdash \hat{e} / a \iff \vdash ||\hat{e}|| : ||a||
\]
\end{lemma}

This lemma captures the fact the analysis performs (at least) two things implicitly: First, typing the term without the annotations, and second, typing the term after the e-marked tagging and untagging operations have been eliminated. Note that we would not be able to prove this property if we used \( a \) instead of \( c \) in the last rules. Next we prove stronger, semantic properties about the analysis.

\begin{proof}
Both directions by simple inductions.
\end{proof}

\section{Semantic Properties of Tag Elimination}

Tag elimination changes the type of a term, and so, necessarily, changes the semantics of the term. Fortunately, it is possible to give a simple and accurate account of this change in semantics using so called wrapper/unwrapper
functions $W, U : B \times A \to \hat{E}$ (which are a bit more general than the classic embedding/projection pair discussed in the next section):

\[
\begin{align*}
W_{b,D} &\equiv \lambda x. x \\
W_{b,V} &\equiv \lambda x. x \\
W_{b,E D} &\equiv \lambda x. E_b x \\
W_{b,F,a} &\equiv \lambda x. F_b (W_{b,a} x) \\
W_{b,a_1 \rightarrow a_2} &\equiv \lambda f. \lambda x. W_{b,a_2} (f(W_{b,a_1} x)) \\
U_{b,D} &\equiv \lambda x. x \\
U_{b,V} &\equiv \lambda x. x \\
U_{b,E D} &\equiv \lambda x. E_b^{-1} x \\
U_{b,F,a} &\equiv \lambda x. U_{b,a}(F_b^{-1} x) \\
U_{b,a_1 \rightarrow a_2} &\equiv \lambda f. \lambda x. U_{b,a_2}(f(W_{b,a_1} x))
\end{align*}
\]

For simplicity, we will write $W_a$ (and similarly $U_a$) for $|W_{b,a}|| = |W_{k,a}||$. The wrapper and unwrapper functions at a given type $a$ can be seen as completely determining a “type-specialization path”.

**Lemma 5 (Wrapper/Unwrapper Types and Annotated Types).**

1. $\vdash W_{k,a}/||a|| \to |a|$  
2. $\vdash W_{e,a}/||a|| \to a$  
3. $\vdash |W_{b,a}| : ||a|| \to |a|$  
4. $\vdash ||W_{k,a}|| : ||a|| \to |a|$  
5. $\vdash ||W_{e,a}|| : ||a|| \to ||a||$  

The unwrapper function has the dual types.

**Proof.** The first two are by simple inductions. The last two come from the basic properties of erasure, and the first two properties. In all cases, we have to establish the properties of the unwrapper function simultaneously. $\square$

**Lemma 6 (Simulating Erasure).** For all $\vdash \hat{e}/a$ and $\vdash \hat{v}/a$ we have

\[ |\hat{e}| \leftrightarrow |\hat{v}| \iff ||\hat{e}|| \leftrightarrow ||\hat{v}|| \]

**Proof.** The forward direction is by a simple induction on the height of the derivation. The backward direction is by an induction on the lexicographic order generated by the height of the derivation and then the size of the term. $\square$

**Corollary 1.** Lemma 6 has a number of useful consequences:

1. For $\vdash \hat{e}_1/a$ and $\vdash \hat{e}_2/a$, we have

\[ |\hat{e}_1| \approx |\hat{e}_2| \iff ||\hat{e}_1|| \approx ||\hat{e}_2|| \]

2. For $\vdash \hat{e}/t$ we have

\[ ||\hat{e}|| \approx |\hat{e}| \equiv |||\hat{e}||| \equiv ||||\hat{e}|||| \]

**Lemma 7 (Projecting Embeddings).** For all $\vdash v : ||a||$

\[ U_a(W_a v) \approx v \]
**Proof.** By induction on the structure of the annotated types. □

For any $\hat{e}$ such that $\vdash \hat{e} : |a|$, the tag elimination transformation $\text{TE}(\hat{e}, a)$ is defined as:

$$\text{TE}(\hat{e}, a) \equiv \begin{cases} ||\hat{e}|| & \vdash \hat{e}/a \\ U_a |\hat{e}| & \text{o.w.} \end{cases}$$

Note that the input to the tag-elimination transformation is an annotated term $e$ and an annotated type $a$. Both the annotations and the annotated type are used to ensure that the transformation is functional. The study of inference techniques for the annotations and the annotated type can alleviate the need for the annotations and the annotated type, but we leave this for future work. Leaving out inference is pragmatically well-motivated, because it is easy for the programmer to provide these inputs.

**Theorem 1 (Extensional Semantics of Tag Elimination).** For all $\vdash \hat{e} : |a|$

$$\text{TE}(\hat{e}, a) \approx U_a |\hat{e}|$$

The proof technique used in a previous study on tag elimination [15] works here.

**Proof.** We only need to prove that for all $\vdash \hat{e}/a$ we have

$$||\hat{e}|| \approx U_a |\hat{e}|$$

This proof proceeds by induction the structure of the annotated type $a$. In the case of $D$ and $V$, the proof comes from the fact that $||\hat{e}|| \approx |\hat{e}|$ when the annotated type $a$ is simply a type. In the case of $E D$ and $F a$, the proof uses the definition of erasure, and the induction hypothesis. The case of arrows is the most interesting. It is done using simulation, extensionality, lifting, the ontoness of both erasure functions, and the second part of Corollary [11] □

5 **Interpreters**

"To explain what interpreters do it is worthwhile to start by discussing the differences between interpreting and translation."

Introduction on web-page of

*Russian Interpreters Co-op (RIC).*

In order to be able to address the issue of Jones-optimality formally and to establish that a certain program is indeed an interpreter, we will need to review some basic issues of encoding and expressibility. Because we are interested in typed interpreters, we will begin by refining our notation and define the sets of typed terms and values as

$$e \in \mathbb{E}_{\Gamma \vdash t} \iff \Gamma \vdash e : t \quad \text{and} \quad v \in \mathbb{V}_{\Gamma \vdash t} \iff \Gamma \vdash v : t$$
And we will write $\mathbb{E}_t$ and $\mathbb{V}_t$ when $\Gamma$ is empty. By proving type preservation on closed terms, we now can give evaluation a finer type $\leftarrow_t \mathbb{E}_t \rightarrow \mathbb{V}_t$.

We define a **first-order datatype** as a type $D$ whose values can be tested for meta-level syntactic equality within the language. That is, for all $v_1, v_2 \in \mathbb{V}_D$, 

$$v_1 \equiv v_2 \iff v_1 \approx v_2$$

Note that, in general, it is desirable that a language have types which do not have this property. In meta-programming settings, it is dangerously easy to thusly “trivialize” observational equivalence for all types $\mathbb{E}$ by proving type preservation on reduction become semantically unsound. The type $D$ in the language presented above is a first-order datatype.

A programming language has **syntactic self-representation** if (1) it has an first-order data type $D$, and 2) there exists a full embedding $\check{\cdot} : \mathbb{E} \rightarrow \mathbb{V}_D$, meaning that:

- $\check{e}$ is defined for all $e$, and
- $\check{\cdot}$ has a left inverse called $\check{-1} : \mathbb{V}_D \rightarrow \mathbb{E}$, that is, $\check{\check{e}} = e$.

The left inverse does not have to be a total function; it just needs to be defined on all elements of the range (image) of the embedding.

For our language, we can define the function and its left-inverse as follows:

\[
\begin{align*}
\check{x} & \equiv x \\
\check{e_1 \ e_2} & \equiv \text{apply(} \check{e_1} \check{e_2} \text{)} \\
\check{\lambda x. e} & \equiv \text{lambda(} \check{x} . \check{e} \text{)} \\
\check{\text{fix } x. e} & \equiv \text{fix(} \check{x} . \check{e} \text{)} \\
\check{s} & \equiv \text{quote.} s \\
\check{\text{car } e} & \equiv \text{car(} \check{e} \text{)} \\
\check{\text{cdr } e} & \equiv \text{cdr(} \check{e} \text{)} \\
\check{\text{cons } e_1 \ e_2} & \equiv \text{cons(} \check{e_1} \check{e_2} \text{)} \\
\check{\text{equal? } e_1 \ e_2} & \equiv \text{equal?(} \check{e_1} \check{e_2} \text{)} \\
\check{\text{if } e_1 \ e_2 \ e_3} & \equiv \text{if(} \check{e_1} \check{e_2} \check{e_3} \text{)} \\
\check{E} & \equiv \text{tagE.} \check{e} \\
\check{E^{-1} e} & \equiv \text{untagE.} \check{e} \\
\check{F} & \equiv \text{tagF.} \check{e} \\
\check{F^{-1} e} & \equiv \text{untagF.} \check{e}
\end{align*}
\]

and,

\[
\begin{align*}
\check{x} & \equiv x \\
\check{\text{apply.} (e_1 \ e_2)} & \equiv \check{e_1} \check{e_2} \\
\check{\text{lambda.} (x \ e)} & \equiv \lambda x. \check{e} \\
\check{\text{fix.} (x \ e)} & \equiv \text{fix(} \check{x} . \check{e} \text{)} \\
\check{\text{quote.} s} & \equiv s \\
\check{\text{car.e}} & \equiv \text{car(} \check{e} \text{)} \\
\check{\text{cdr.e}} & \equiv \text{cdr(} \check{e} \text{)} \\
\check{\text{cons.e.} (e_1 \ e_2)} & \equiv \check{\text{cons.e.} (e_1 \ e_2)} \\
\check{\text{equal?.} (e_1 \ e_2)} & \equiv \check{\text{equal?.} (e_1 \ e_2)} \\
\check{\text{if.e.} (e_1 \ e_2 \ e_3)} & \equiv \check{\text{if.e.} (e_1 \ e_2 \ e_3)} \\
\check{E.e} & \equiv \check{E.e} \\
\check{E^{-1} e} & \equiv \check{E^{-1} e} \\
\check{F.e} & \equiv \check{F.e} \\
\check{F^{-1} e} & \equiv \check{F^{-1} e}
\end{align*}
\]

It is easy to see that $\check{e} \equiv e$. Such encoding/decoding pairs have sometimes been called **reify** and **reflect**. This terminology was promoted by Brian Cantwell Smith [12]. Reify provides us with a way of “materializing” or “representing”...
terms within the language, and reflect provides us with a way of interpreting an internal representation back into a (meta-level) term. Note that all these functions exist at the meta-level, and that expressing them within the language requires first defining them at the meta-level.

As with evaluation, we will be more interested in the “subject-typed” versions of these functions: $\tau^t_t : \mathbb{E}_t \rightarrow \mathbb{V}_t$ and $\llcorner t : \mathbb{V}_t \rightarrow \mathbb{E}_t$, where the first one is achieved by restricting the input to be well-typed, and the second by restricting the output to being well-typed.

With syntactic representation in hand, it is tempting to view interpreters as a program (call it direct) expressing the following function:

$$\llcorner \tau \hookrightarrow \tau : \mathbb{V}_t \rightarrow \mathbb{V}_t$$

Because such a function produces a value of the same (subject) type as the (subject) term being interpreted, we will call them direct interpreters. But it turns out that expressing such an interpreter in a statically typed programming language (such as the one at hand) is a rather subtle matter. In fact, it is only recently that some work on programming type-indexed values in ML [14] has given a hint of how such a function can be expressed. But even then, it is known that we can express such an “interpreter” for each type, but it is not known that there is one term that we can call the interpreter and that would work for all types.

5.1 Expressibility and Admissibility of Encoding/Decoding

While the encoding and decoding function presented above would generally be enough for expressing an interpreter in an untyped setting, they are generally not enough in a typed setting. To clarify this point, we will analyze the expressibility of these functions and of interpreters.

A partial (meta-level) function $f : \mathbb{V}_{t_1} \rightarrow \mathbb{V}_{t_2}$ is expressed by a term $e_f \in \mathbb{V}_{t_1 \rightarrow t_2}$ when for all $v \in \mathbb{V}_{t_1}$

$$e_f v \approx f(v).$$

As a simple example, for any $t$, the function $id : \mathbb{V}_t \rightarrow \mathbb{V}_t$ is expressed by the term $e_{id} \equiv \lambda x.x \in \mathbb{V}_{t \rightarrow t}$. In contrast, any function that distinguishes between the terms $\lambda x. (\lambda y.y)x$ and $\lambda x.x$ would not be expressible. A partial meta-level function $f : \mathbb{V}_{t_1} \rightarrow \mathbb{V}_{t_2}$ is admissible when for all $v_1, v_2 \in \mathbb{V}_{t_1}$ such that $v_1 \approx v_2$ if $f(v_1)$ is defined then 1) $f(v_2)$ is defined, and 2) $f(v_1) \approx f(v_2)$. Expressible functions are admissible, but not necessarily the converse. Thus, admissibility helps in establishing negative statements on what can be expressed.

As is, the two untyped functions described above cannot be expressed in our language; in both cases, they don’t have the right type: at least they need to be restricted to values in both the domain and the co-domain.

---

8 Here we omit the formal definition of “expresses” for reasons of space.
If we restrict the decoding function to values and its result to values of type $V \to V$, we get a function of type $V_D \to V_{V \to V}$. This encoding function is admissible (in fact, even though we don’t prove it, we expect that it is expressible). Because one of the main things that we generally want to do with expressions is to evaluate them after decoding them, a decoding function restricted to values almost (but not quite) models an interpreter.

However, if we restrict the encoding function to values, and then further restrict it to values of some type, say, type $V \to V$, we get a function of type $V_{V \to V} \to V_D$. This function distinguishes between operationally equivalent terms, therefore, it is not even admissible.

Because of the subtleties involved in expressing a direct interpreter, a more commonly used technique for implementing interpreters involves the use of a universal datatype. We define a universal datatype as a type $V$ that allows us to simulate values of any type by a value of one (universal) type. We can formalize the notion of simulation concisely as follows: There must exist a universal embedding function $w_t : \forall t \to \forall V$ such that:

$$v_1 \approx v_2 \iff w_t(v_1) \approx w_t(v_2)$$

We can establish that the datatype $V$ in our programming language is a universal datatype by using a family of terms $E_t$ and $P_t$ and showing that the latter is a left-inverse of the former:

$$
E_D \equiv \lambda x. E x \\
E_V \equiv \lambda x. x \\
E_{t_1 \to t_2} \equiv \lambda f. \lambda x. E_{t_2}(f(P_{t_1} x)) \\
P_D \equiv \lambda x. E^{-1} x \\
P_V \equiv \lambda x. x \\
P_{t_1 \to t_2} \equiv \lambda f. \lambda x. P_{t_2}(F^{-1} f(E_{t_1} x))
$$

And it is easy to show that $P_t(E_t v) \approx v$.

**Lemma 8 (Projecting Embeddings).** $P_t(E_t v) \approx v$

**Proof.** By induction on the structure of the types. □

**Remark 1.** Note that the fact that we don’t need to apply the induction hypothesis in the case of $V$ is essential for the ability to do the proof by induction on the structure of types.

Such a universal datatype plays a crucial role in allowing us to express simple interpreters in non-dependently typed programming languages. In particular, they allow us to implement typed interpreters by what we will call an indirect interpreter. A term is an indirect interpreter (call it indirect) if it expresses the function:

$$(\downarrow_t ; \leftarrow_t ; w_t) : \forall D \to \forall V$$

While this shift from direct to indirect interpreters makes writing interpreters easier, it also introduces the very overhead that the tag elimination transformation will need to remove. In our Scheme-based implementation the self-interpreter is essentially as follows:

9 Unfortunately, space does not allow us to give all the details here.
(fix newenv-eval (lambda env (fix myeval (lambda e
(if (atom? e) (app env e)
(if (equal? (car e) (quote lambda)) (.tagF. (lambda x (app (app newenv-eval
(l lambda y (if (equal? y (car (cdr e))) x (app env y)))) (car (cdr (cdr e))))))))
(if (equal? (car e) (quote app)) (app (.untagF.
(app myeval (car (cdr e))))) (app myeval (car (cdr (cdr e)))))
(if (equal? (car e) (quote tagF)) (tagF (.untagF. (app myeval (car (cdr e))))))
...)

Where, for example, tagF is \(F_k\) and .tagF. is \(F_e\). We will define our typed self-interpreter \(\text{tsi}\) to be the term above specialized (by simple application) to the empty environment.

It is folklore that \(\text{tsi}\) is an indirect interpreter and we do not prove it here.

6 Staged Interpreters and Translation

We mentioned in the introduction that a staged interpreter can be viewed simply as a translation. This is a subtle shift in perspective. In particular, the only requirement on interpreters is that they yield “the right value” from a program. Often, the straight-forward implementation of interpreters (in both CBN and CBV programming languages) tends to have a pragmatic disadvantage: They simply do not ensure a clean separation between the various “stages” of computing “the right value” of an expression. In particular, straight-forward implementations of interpreters tend to repeatedly traverse the expression being interpreted. Ideally, one would like this traversal to be done once and for all. In general, achieving this kind of separation gives rise to the need for using two- and multi-level languages.

“Staged interpreters”, therefore, are not any composition of the functions described above, but rather, a particular implementation of this composition that behaves in a certain manner. Because CBN and CBV functional languages cannot force evaluation under lambda, they are thought to be insufficient for expressing staging. Nevertheless, the result of a staged interpreter (which is a term in the target language corresponding to the given term in the subject language) is expressible in the language. Furthermore, the result of a staged interpreter is also observationally equivalent to the result of an interpreter. These facts imply that, while staged interpreters are not known to be expressible in a language such as the one we are studying in this paper, they are still admissible.

Pragmatic experience with staged interpreters suggests that their input-output behavior can be modeled rather straight-forwardly as a translator. For simplicity, it is enough to focus on a self-interpreter: an interpreter written in the same language it interprets. When staged, a self-interpreter translates terms in one language into terms in the same language. Note however, that for typing reasons, the resulting translation is not the identity. Rather, it is the following function:
\[ E(x) \equiv x \]
\[ E(e_1, e_2) \equiv (E_{e_1}^{-1} E(e_1)) E(e_2) \]
\[ E(\lambda x. e) \equiv F_e \lambda x. E(e) \]
\[ E(\text{fix } x. e) \equiv \text{fix } x. E(e) \]
\[ E('s) \equiv E e 's \]
\[ E(o e) \equiv E e o (E_{e_1}^{-1} E(e)) \]
\[ E(\text{if } e_1, e_2, e_3) \equiv \text{if } (E_{e_1}^{-1} e) E(e_2) E(e_3) \]
\[ E(D) \equiv E D, \quad E(V) \equiv V, \quad E(t_1 \rightarrow t_2) \equiv F (E(t_1) \rightarrow E(t_2)) \]

One can show that \(|E(e)| \approx (\text{tsi } e)^{\gamma} \). It is reasonable to expect that a staged \text{tsi} produces \(|E(e)|\) when applied to \(^{\gamma} e\), and our implementation confirms it.

The annotated type of the result of translating a term of type \(t\) is defined as follows:

\[ E(D) \equiv E D, \quad E(V) \equiv V, \quad E(t_1 \rightarrow t_2) \equiv F (E(t_1) \rightarrow E(t_2)) \]

The idempotence of the translation on \(V\) is essential for being able to do the various proofs that are carried out by induction on the structure of the types.

**Lemma 9 (Soundness (and Full-Abstraction) of Translation).**

\[ e_1 \approx e_2 \iff |E(e_1)| \approx |E(e_2)| \]

**Proof.** Proved by showing that \(E(e) \approx W_e\), and Projecting Embeddings lemma. \( \square \)

**Lemma 10 (Well-Typed Terms “Go Through”).**

1. \( \Gamma \vdash e : t \implies E(\Gamma) \vdash E(e)/E(t) \)
2. \( ||E(e)|| \equiv e \)

**Proof.** Both by a simple induction on the structure of \(e\). \( \square \)

The first part of this lemma means that running the staged interpreter on a well-typed subject program yields a term that passes the tag elimination analysis. The second part means that erasing the operations marked by \(e\) yields back precisely the term that we started with.

Note further that, using the second part, we can strengthen the first part of this lemma to be:

\[ \Gamma \vdash e : t \iff E(\Gamma) \vdash E(e)/E(t) \]

This statement means that applying the tag elimination analysis to the result of a staged self-interpreter is exactly the same as type-checking the term being interpreted. This is probably the most accurate characterization of the strength of the idea of tag-elimination.

\footnote{Now we can see how \(U\) and \(W\) generalize \(P\) and \(E\). For example, \(W_{\varepsilon(t)} = E_t\).}
7 Jones-Optimal Specialization

At this point, we have presented a variety of results that indicate that tag elimination has a useful application in the context of self-application of a specific typed programming language, and would therefore be useful in improving the effectiveness of traditional partial evaluators in staging many interesting interpreters written in this language. Now we turn to addressing the long-standing open problem of Jones-optimality, formally. A function $PE$ is a partial evaluator if, for all closed $e_1$ and $e_2$:

$$PE(e_1, e_2) \approx e_1 e_2$$

A partial evaluator is partially-correct if it is a partial function satisfying the above equation, when defined. Note that we require $e_1$ and $e_2$ to be closed only for simplicity, as partial evaluators are syntactic operations and therefore must deal with free variables anyway. A function $tPE$ is a typed partial evaluator if

$$\vdash e_1 e_2 : a \implies tPE(e_1, e_2, a) \approx U_a (e_1 e_2)$$

A partial evaluator is partially-correct if it is a partial function satisfying the above equation, when defined. This definition of a typed partial evaluator is motivated by the definition of an self-interpreter in a typed programming language. A self-interpreter $si$ is a term such that:

$$si \vdash e \approx e$$

A typed self-interpreter $tsi$ is a term such that:

$$\vdash e : t \implies tsi \vdash e \approx w_t e$$

where $w_t$ is a universal embedding function. Now we can recapitulate the definition of Jones-optimality [5]. A partial evaluator $PE$ is Jones-optimal with respect to an untyped self-interpreter $si$ when for all $\vdash e : t$ we have

$$PE(si, \vdash e) \equiv e$$

Again motivated by the role that a universal datatype plays in typed interpreters, we generalize the definition of Jones-optimality to the typed setting as follows: A typed partial evaluator $tPE$ is said to be Jones-optimal with respect to a typed self-interpreter $tsi$ when for all $\vdash e : t$:

$$tPE(tsi, \vdash e, \mathcal{E}(t)) \equiv e$$

Theorem 2 (Main).

1. Whenever $PE(\_ , \_)$ is a (partially) correct partial evaluator, $TE(PE(\_ , \_), \_)$ is a (partially) correct typed partial evaluator, furthermore
2. Whenever, for all \( e \) it is the case that \( \text{PE}(\text{esi}, e^\top) \equiv \varepsilon(e) \), then \( \text{TE}(\text{PE}(\_, \_), \_) \) is Jones-optimal.

**Proof.** For the first part, all we need is to follow a simple sequence of semantic equalities:

\[
\begin{align*}
\text{TE}(\text{PE}(e_1, e_2), a) & \text{ by extensional semantics of } \text{TE} \\
& \approx U_a(\text{PE}(e_1, e_2)) \text{ by definition of a PE} \\
& \approx U_a(e_1 e_2)
\end{align*}
\]

and we have satisfied the definition of a tPE.

For the second part, we only have to follow a simple sequence of syntactic equalities:

\[
\begin{align*}
\text{TE}(\text{PE}(\text{esi}, e^\top), \varepsilon(t)) & \text{ by assumption} \\
& \equiv \text{TE}(\varepsilon(e), \varepsilon(t)) \text{ by Lemma 10.1, } \vdash \varepsilon(e)/\varepsilon(t) \text{ so TE “succeeds”} \\
& \equiv \|\varepsilon(e)\| \text{ and by Lemma 10.2} \\
& \equiv e
\end{align*}
\]

and we have satisfied the definition of typed Jones-optimality.

We have built an implementation that supports this result.

## 8 Conclusions and Future Work

In this paper, we have presented the theoretical results showing how Jones-optimality is achieved using tag elimination. We have also implemented a system based on the analysis presented here (in Scheme), and it has validated our theoretical results. The analysis we presented here contains technical improvements over the original proposal in that it uses a simpler judgment. The main reason for this simplicity is that we exploit information about well-formedness of annotated types in the judgment. However, it is also more specialized than the original analysis, which is parametric over an arbitrary datatype that we might want to eliminate.

The moral of the present work is that there is a practical solution to the problem of Jones-optimality which can be attained through some simple annotations by the user. There is evidence that the annotations may not be necessary in practice. Makholm implemented a variant of tag elimination for a first-order language whose type structure is different than that of the language we use here. In this implementation, the analog of our e and k annotations are inferred automatically by the tag eliminator instead of being embedded in the staged interpreter. In principle, it seems that such inference in the setting presented in this paper should be decidable although not necessarily efficient. Whether or

---

11 The system can be downloaded from [http://www.diku.dk/~makholm/teal.tar.gz](http://www.diku.dk/~makholm/teal.tar.gz).

12 In fact, in this paper, we have only talked about well-formed annotated types. There is a general way to go from the original definition of well-formedness to the kind of presentation given here, but this is beyond the limits of space available here.
not efficient and practical inference will scale to the higher-order setting is not known. The work on dynamic typing may help establish such a result formally. Combinations of wrapper and unwrapper functions provide natural mechanisms for a notion of subsumption or subtyping that can be used to provide an analog of soft-typing. Finally, we hope to generalize this work to richer settings with state and polymorphism.

Acknowledgements: We are very grateful to Neil Jones, who played an active part in the early stages of this work, and offered valuable comments thereafter. We also enjoyed many useful discussions with Makoto Takayama and Peter Thiemann. We received many useful comments on earlier drafts from Christopher League, Henrik Nilsson, Stefan Monnier, Carsten Schürmann, Valery Trifonov, and an anonymous referee.

References


Bottom-Up Deduction with Deletion and Priorities

Harald Ganzinger

Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85, D-66123 Saarbrücken, Germany
hg@mpi-sb.mpg.de

McAllester has recently shown that the running time of a bottom-up logic program can be bounded by the number of “prefix firings” of its inference rules. This theorem allows one to view a logic program as an algorithm whose running time is given by the number of prefix firings of the rules. Although pure logic programs under prefix firing running time are adequate for many algorithms, many other algorithms seem to lie outside of this framework.

Here we extend the concept of inference systems to one in which atoms can also be deleted in the course of application of an inference rule. Deletion makes the behavior of the algorithm nondeterministic. For example, consider the following rules with deletion where the marking \([\ldots]\) means that the premise is to be deleted as soon as the rule is applied.

\[
P \Rightarrow Q \quad [Q] \Rightarrow S \quad [Q] \Rightarrow W
\]

Suppose the initial data base contains only \(P\). The first rule fires, adding the assertion \(Q\). Now either the second or third rule can fire. Since each of these rules deletes \(Q\), once one of them fires, the other is blocked. Hence the final data base is either \(\{P, S\}\) or \(\{P, W\}\), nondeterministically. When viewing rules with deletions as algorithms, this nondeterminism is viewed as “don’t care” nondeterminism—the choices are made arbitrarily and irrevocably. On the other hand, the time needed to compute a deductive closure of a data base might crucially depend on which execution path is chosen. To gain control over this phenomenon, we allow priorities to be attached to the rules, specifying the order in which rules should fire.

We prove a general meta-complexity theorem for such inference systems. This theorem can be viewed as establishing a prefix-firing notion of running time for programs with deletion and priorities. We show that under this notion of running time one can give efficient logic-program implementations of union-find and other algorithms, such as congruence closure, which depend on union-find.

Logic programs with deletion and priorities, although more complex than pure logic programs, still seem simpler and more easily analyzed than classical pseudo-code based on iteration and recursion. As an example we give a high-level

* The results reported here were obtained in collaboration with David McAllester, AT&T Labs-Research.
formulation of Dijkstra’s shortest path algorithm. This algorithm requires priorities for rule instances rather than rule schemas. A problem requiring priorities only on the level of rule schemas is ground Horn satisfiability in the presence of equality. We give an algorithm that, to our knowledge, is the first $O(n \log n)$ algorithm for this problem.

References

Author Index

Chin, Wei-Ngan, 2
Fischbach, Adam, 25
Ganzinger, Harald, 276
Glynn, Kevin, 39
Gustavsson, Jörgen, 63
Hannan, John, 25
Hughes, John, 257
Jones, Neil D., 1
Khoo, Siau-Cheng, 2
Lawall, Julia L., 84
Makholm, Henning, 257
Massé, Damien, 103
Mastroeni, Isabella, 117
Masuhara, Hidehiko, 138
Miné, Antoine, 155
de Moor, Oege, 218
Schultz, Ulrik P., 173
Secher, Jens Peter, 198
Sittampalam, Ganesh, 218
Søndergaard, Harald, 39
Stuckey, Peter J., 39
Sullivan, Gregory T., 238
Sulzmann, Martin, 39
Svenningsson, Josef, 63
Taha, Walid, 257
Xu, Dana N., 2
Yonezawa, Akinori, 138